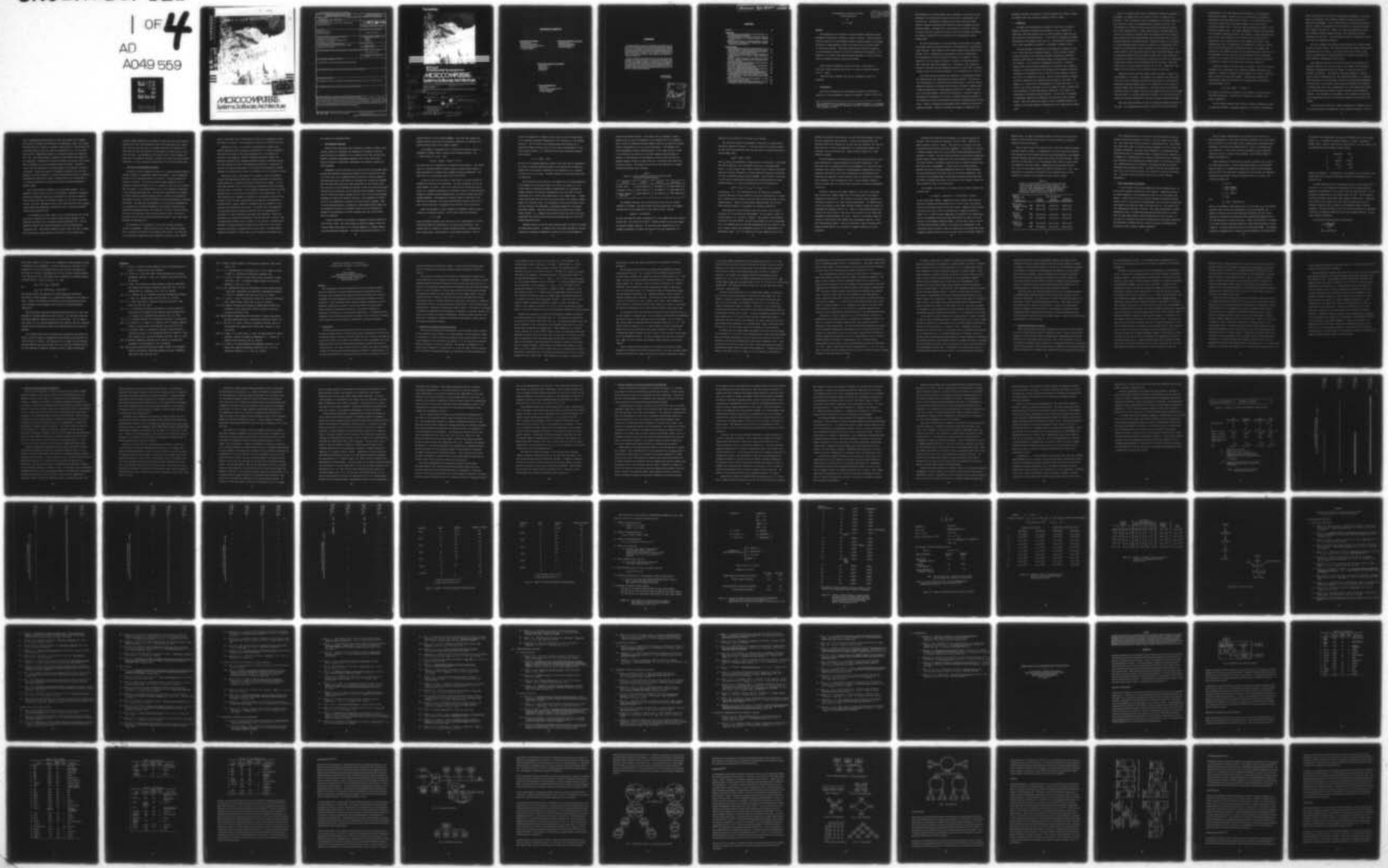
AD-A049 559    COLORADO STATE UNIV FORT COLLINS DEPT OF ELECTRICAL --ETC F/G 9/2
PROCEEDINGS OF THE ANNUAL ROCKY MOUNTAIN SYMPOSIUM ON MICROCOMP--ETC(U)
SEP 77 M ANDREWS, S MCCORMICK    AFOSR-77-3367

UNCLASSIFIED    AFOSR-TR-78-0017    NL

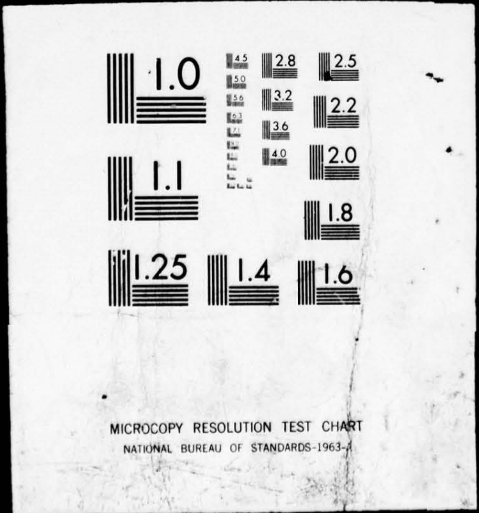
1 OF 4
AD
A049 559



SIFIED

1 OF 4

AD
A049 559


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER AFOSR-TR- 78-0017 ✓ | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) MICROCOMPUTERS: SYSTEMS, SOFTWARE, ARCHITECTURE | | 5. TYPE OF REPORT & PERIOD COVERED Final AUG. 31 - Sept 2, 1977 ~~1 Jun 77 - 31 May 78~~ |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s) Michael Andrews, Steve McCormick | | 8. CONTRACT OR GRANT NUMBER(s) AFOSR 77-3367 New |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS Colorado State University, Department of Electrical Engineering ✓, Ft Collins, CO 80523 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM, Bolling AFB DC 20332 | | 12. REPORT DATE Sep 2, 1977 |
| | | 13. NUMBER OF PAGES 310 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) | | 15. SECURITY CLASS. (of this report) UNCLASSIFIED |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Proceedings of the First Annual Rocky Mountain Symposium, IEEE, Printed in USA

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Rocky Mountain symposia on microcomputers are organized as a forum for presentation and discussion of basic research in the field of microcomputers. The aim is to encourage a broad base of interaction between the industrial, governmental, and academic communities, thus providing guidance and feedback for the directions of basic research and for the effective implementation of research achievements.

Proceedings

(6) Proceedings of the

# 1st Annual Rocky Mountain Symposium on MICROCOMPUTERS: Systems, Software, Architecture

(1st)

(11) 2 Sep 77

(12) 316p.

DDC
RECEIVED
FEB 3 1978
A

August 31 – September 2, 1977
Fort Collins, Colorado

(15)

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is
approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

AFOSR-77-3367

Sponsors

Air Force Office of Scientific Research
Army Research Office

(16) 2304

Contributing Organizations

(10) Michael Andrews
Steve McCormick

Colorado State University

American Society of Engineering Education

IEEE Computer Society

(17) A2

(9) Final rept. 1 Jun 77 – 31 May 78,

(18) AFOSR

Additional copies available from:

IEEE Computer Society
5855 Naples Plaza, Suite 301
Long Beach, CA 90803

(19) TR-78-0017

406 434

## ORGANIZING COMMITTEE

**CONFERENCE CHAIRMAN**
Michael Andrews
Microprocessor Laboratory
Department of Electrical Engineering
Colorado State University
Ft. Collins, CO 80523

**TECHNICAL PROGRAM CHAIRMAN**
Stephen McCormick
Department of Mathematics
Colorado State University
Ft. Collins, CO 80523

**SUPPORTING COMMITTEE MEMBERS**
Gearold Johnson
Gerald Taylor
Michael Tyndall
Rodney Goke

**LOCAL ARRANGEMENTS**
Carolyn Frye
Office of Conferences and Institutes
Colorado State University
Ft. Collins, CO 80523

# FOREWORD

The Rocky Mountain symposia on microcomputers are organized as a forum for presentation and discussion of basic research in the field of microcomputers. The aim is to encourage a broad base of interaction between the industrial, governmental, and academic communities, thus providing guidance and feedback for the directions of basic research and for the effective implementation of research achievements. Promotion of such cooperation and stimulation of interest in these objectives should have major impact on the future of microcomputers and their uses.

From the standpoint of these objectives, the First Annual Rocky Mountain Symposium on Microcomputers has proved successful. Participation from government and academia is manifest in the sponsorship of this symposium and in the research papers published in the following pages. Participation by the industrial community, toward whom much of the material is directed, is especially evident in the audience and the special panel discussion. The continuation of dialogue between these communities must play a critical role in the future of microcomputer development.

**Michael Andrews**
**Steve McCormick**

NOT
Preceding Page Blank - Filmed

## CONTENTS

# The Challenge in Numerical Software for Microcomputers

by

W. J. Cody*

## Abstract

Microcomputers are now capable of serious numerical computation using programmed floating-point arithmetic and Basic compilers. Unless numerical software designers for these machines exploit experience gained in providing software for larger machines, history will repeat with the initial spread of treacherous software. This paper discusses good software, especially for the elementary functions, in terms of reliability and robustness. The emphasis is on insight rather than detailed algorithms, to show why certain things are important and how they may be achieved.

*"What history and experience teach us is this -- that people . . . never have learned anything from history, or acted on principles deduced from it"* - Hegel

*"Those who cannot remember the past are condemned to repeat it"* - Santayana

## 1. Introduction

We are now witnessing in the microcomputer industry an accelerated repetition of the development of computers in general. Within a short time

*Work performed under the auspices of the U.S. Energy Research and Development Administration in the Applied Mathematics Division, Argonne National Laboratory, Argonne, IL 60439.

microcomputers have achieved memory size, wordlength, instruction sets and peripherals with capabilities rivaling those found on minicomputers just a few years ago. Microcomputer software now includes operating systems, algebraic language interpreters and compilers, graphics packages and text-editing programs. Although most of this software is primitive, within a few years there may be almost no practical distinction between the hardware and software capabilities of microcomputers and minicomputers, except perhaps speed.

One discouraging aspect of this development is that much of the emerging software ignores the lessons learned in the development of software for larger machines. (New software for the larger machines often ignores these lessons as well, reaffirming the observations of Hegel and Santayana.) Once disseminated, inferior software is difficult to eradicate. This is especially true for numerical software -- witness the lingering death of IBM's Scientific Subroutine Package [11]. Fortunately, little numerical software is available for microcomputers and there is still time to produce a decent product for the first generation. The challenge is to do the job.

This paper discusses problems and techniques in preparing good numerical software based upon experiences with larger computers. The intent in this presentation is to provide insight and motivation, to show why certain things are important and indicate how they may be achieved. Detailed algorithms and descriptions of good numerical software for a variety of tasks are to be found elsewhere, and are not included here. Section 2 begins by considering the background of such work, including limitations imposed on the software by current and expected microcomputer hardware/software environments. Section 3 introduces and discusses certain desirable attributes of numerical software. Section 4 discusses the design of software for the elementary functions, and Section 5 briefly discusses other items of numerical software that will probably be among the first to appear.

## 2. Background

We assume that floating-point arithmetic of some form is available. Numerical software can and does exist without it, e.g., software for numerically controlled machine tools, but most scientific computation and data reduction is best done with it. To the author's knowledge no hardware floating-point microcomputer CPUs exist yet. Floating-point operations are still either programmed or provided on a peripheral arithmetic device.

The most important consideration in floating-point arithmetic is that it be "clean" and free of anomalous behavior. Clean arithmetic properly rounds the results of the four basic arithmetic operations, generating necessary guard digits in intermediate stages of the operations to protect the rounding. It is also free of mathematical *surprises* -- *numerical* behavior that deviates without warning from the expected norm. Anomalous behavior is often associated with the fringes of the arithmetic system and considered to be ignorable by all but the most finicky numerical analysts, but this is not necessarily the case. Some of the most insidious examples can affect computations far removed from the fringes of the system. For example, on a certain line of minicomputers it is often true that A + A is correct, but 2.0*A is incorrect by up to 15 times the normal rounding error. This discrepancy, as with many anomalous behaviors, is the result of a combination of unfortunate design decisions. In this case the arithmetic is hexadecimal coupled with rounding by truncation and a lack of guard digits. When such combinations appear on short wordlength machines, the results of even simple numerical computations of any length must be suspect [16].

The set of computers with clean floating-point arithmetic, hardwired or programmed, is probably empty even though it is easy to design such an arithmetic. When floating-point must be programmed it is clearly harder and more expensive to write a clean package than a slightly dirty one. The clean package also probably occupies more space and executes more slowly. All of this translates into a visible expense that the package author and user are each anxious to eliminate under the assumption that the shortcuts taken introduce only minor perturbations from the expected norm. The same considerations apply to hardware design. Engineers attempt to cut expenses by eliminating guard digits and proper rounding, or by substituting clever circuitry to perform almost correct arithmetic in more efficient ways. A classic example is the large scale machine which, among other things, treats floating-point numbers with the smallest exponent and a legal significand as nonzero for addition and subtraction operations, but which for engineering reasons treats them as zero for multiplication and divisions. Thus there exist nonzero X on this machine for which 1.0*X is zero. Admittedly these numbers lie on the fringe of the arithmetic system in this case.

Anomalies can also arise when honest efforts are made to improve the characteristics of a design. For example, some computers carry more precision in the active arithmetic registers than they carry in stored numbers, essentially limiting roundoff error in any computation to that incurred in storing a number away and recovering it. But then there are two possibly different values of a number, the one in the registers and the one in storage, which can lead to a situation where X-X is nonzero. This has obvious implications for testing convergence of iterative processes, among other things.

Even with clean arithmetic the job of writing good numerical software is not easy. All computations must still be carried out over that discrete bounded subset of the real number system dictated by the floating-point representation scheme. Usually the set is closed under the arithmetic operations by including underflow and overflow. But the arithmetic operators may not always return the closest element to the "true" result, introducing roundoff error. There are other peculiarities traceable to the arithmetic system as well. The common hexadecimal floating-point representation introduces "wobbling precision," for instance, in which there may be as many as three fewer significant bits, almost one less decimal place of significance, in the representation of some numbers than others. This phenomenon may force a reorganization of computation to avoid poor significance in the representation of intermediate results. For example, one algorithm for the tangent function uses the fractional part of X*(4/pi). But the hexadecimal constant 4/pi has less significance than the hexadecimal constant pi/4, and almost one extra decimal place of accuracy can sometimes be obtained by using the fractional part of X/(pi/4) instead [6].

Supportive software is another factor to consider. The cost of unclean arithmetic can be compounded by poor algebraic language interpreters and compilers. Consider the standard Fortran compilers on the line of machines which confuses zero and nonzero numbers. Because these compilers use floating-point addition and subtraction in making logical comparisons, a Fortran statement such as

```
IF (X .NE. 0.0E0)  Y = 1.0E0 / X
```

can trigger an interrupt for division by zero despite the obvious effort to avoid that possibility. (The "fix" is to replace X by 1.0E0*X in the logical expression.)

As with hardware designs, honest efforts to improve compilers can lead to unexpected problems. Optimizing compilers, for instance, often greatly improve inefficient code written by inexperienced programmers; on the other hand, if unrestrained, they sometimes undermine careful code written by experienced programmers. It is a common practice on machines without guard digits to rewrite the expression X-1.0 as (X-0.5)-0.5 to preserve accuracy for X slightly less than 1.0. There is at least one compiler which helpfully combines constants whenever possible, thus replacing the second expression with the first. This optimization cannot be suppressed; it must be avoided by programming subterfuge.

The list of hardware and software related pitfalls is a long one which we will not pursue further. The interested reader may consult more detailed discussions such as (3,10,13). The point we have tried to make is that there are peculiarities in existing hardware and supportive software that affect numerical computations in subtle ways.

Good numerical software can be written using unclean arithmetic and imperfect compilers; that is not an issue. Any anomaly can be avoided by appropriate programming provided the anomaly is known and provided its avoidance is important. The problem is that known anomalies constantly inspire evasive programming action, and unsuspected anomalies, or anomalies whose importance is not anticipated, frustrate even experienced programmers. The money saved in the design of less-than-clean arithmetics and compilers is expended many times over in attempting to write self-immunizing software or in naive reliance upon numerical results obtained from non-immunized software. This tremendous cost is hidden. Clean arithmetic and good compilers simply make the job of providing good numerical software easier and cheaper.

For the record, we describe a simple floating-point arithmetic that we feel would facilitate the preparation of good numerical software. We start with a sign-magnitude representation so that every number can be negated, and a binary or decimal, but definitely not hexadecimal, radix. The primary working precision is fourteen to eighteen decimal places of significance with an exponent range at least ten times the significance, i.e., if the arithmetic carries S significant digits it can represent numbers with exponents up to 10S. The exponent range is balanced so that almost every number can be reciprocated. Underflow and overflow interrupts are precise, with replacement of underflow by zero and continued execution possible. Rounding is "clean" with the equivalent of two guard digits, and rounding by truncation is optional. Finally the results of arithmetic operations carry the same precision as stored quantities. A double working precision with a compatible representation would also be useful, although there is an inconsistency in our design philosophy when we double the precision without modifying the exponent range.

Such a floating-point arithmetic is not available anywhere. It is possible to achieve much of this on a microcomputer, although the suggested precision and exponent range may be unrealistic for this type of machine. Less precision and range is acceptable, however, if the rest of the design is good. The worst combination would be short significand and short exponent range coupled with hexadecimal representation and a truncating arithmetic without guard characters [16].

The arithmetic we are most likely to see on microcomputers for the next few years is exemplified by the software floating-point package for the INTEL 8008 and 8080 in use at Lawrence Livermore Laboratory [17], and the modified calculator chip recently announced by National Semiconductor Corporation [18]. The software package is binary, while the chip is decimal. Each provides about eight decimals of precision, but the software only provides decimal exponents up to roughly 20, while the chip carries decimal exponents up to 99. Each provides the four basic operations plus a square root, with the chip from four to ten times slower than the software. The author does not know the details of the arithmetic in either case, but suspects the worst. The importance of the chip is that it also provides the usual array of elementary functions, constants and conversions found on hand calculators. We will discuss the implications of that in more detail later.

## 3. Attributes of Good Numerical Software

We define an item of numerical software as a running documented computer program available in a particular computer environment. This distinguishes software from an algorithm printed in a journal or a numerical method described in a textbook. The item of software probably implements such an algorithm or numerical method, but it is a separate entity not to be confused with the others. It has characteristics all its own completely independent of the underlying algorithm [20]. The same basic algorithm may be imbedded in several different programs differing widely in details of organization and numerical behavior, therefore differing in those characteristics affecting performance. The important new ingredient in the successful implementation of an algorithm is a detailed knowledge of the arithmetic system and the supportive programming systems to be exploited. We saw in the last section that there are times when 2.0*X is best calculated as X+X. Attention to subtleties such as this may make the difference between a very useful implementation and something less.

Our discussion will concentrate on only those software attributes related to performance. Ideally we would like to have numerical software that is accurate and efficient, and resilient under misuse. We would like to have numerical programs that can be trusted to accept our data, operate upon it, and either return valid numerical results or an explanation of why they cannot be obtained. Such programs are described as reliable and robust.

Reliability refers to the ability of the program to perform a well-defined computation both accurately and efficiently. Of course, reliability starts with the choice of the proper algorithm, but it is primarily an attribute of the software. Reliable software successfully handles the problem set defined by the underlying numerical analysis, realizing accuracy over that problem set close to the theoretical prediction. Reliability is relative, and improper appreciation of the computer environment may degrade reliability by restriction of either the problem set or the obtainable accuracy.

Robustness refers to the ability of a program to avoid or gracefully recover from computational difficulties without unnecessary interruption of program execution. Consider the problem of underflow for example. In most cases when underflow occurs an error message is generated and then execution continues with a zero result. If the underflow significantly alters the final computed result it is destructive; otherwise it is non-destructive. The usual underflow error message is annoying to a user because it does not tell him what he wants to know, namely whether the underflow was destructive or non-destructive. Robust software is completely free of underflow, so the question of its importance never arises. To achieve this, the computation is restructured wherever possible, consistent with the requirements for reliability, to avoid expressions that might cause underflow. Should the possibility of underflow be unavoidable, tests are made for it ahead of time and appropriate remedial action is taken if it is detected. Non-destructive underflow is quietly bypassed and execution continues with a zero result; destructive underflow is bypassed with an error return, including precise diagnostic information. Other types of error conditions that might arise are treated in an analogous manner.

## 4. The Elementary Functions

Among the first important items of numerical software to appear in any computer system are subprograms for elementary and algebraic functions. These functions are considered to be so important that they are often predefined in algebraic programming languages such as Basic and Fortran. Special libraries of appropriate subprograms then accompany compilers for these languages.

Fundamental as these functions are, we will see that the programs implementing them sometimes contain gross blunders that go undetected for years. It is still rare for an elementary function library to contain programs of uniformly high quality even though the techniques for writing such programs have been known and practiced by some individuals for over fifteen years. Until now most of this knowledge has appeared in small bits and pieces widely scattered in the literature, and the average system programmer assigned the task of writing the library programs has had to rely on his own often meager knowledge of calculus and numerical analysis. This background is insufficient for the preparation of reliable and robust function programs. Much of the following discussion is based upon a forthcoming software manual for the elementary functions [7] designed to assist just such systems programmers in the preparation of better programs than they could probably write by themselves.

A typical algorithm for evaluating an elementary function consists of three distinct steps. The first accepts an arbitrary argument within the function domain and reduces it to a related argument in a primary domain, plus some additional parameters. The second evaluates the function, or a related function, for the reduced argument. The third then combines the computed function value with the additional parameters to reconstruct the desired function value of the original argument.

As an example, an algorithm for the evaluation of sine(x), where x is expressed in radians, might be based on the following analysis. Let $x = N*pi+f$ where $|f| \leq pi/2$. Then

$$\text{sine}(x) = \text{sign}(x) * \text{sine}(f) * (-1)^N .$$

The first step in the algorithm is to determine N and f given x. The second step is to calculate sine(f), probably using a truncated Taylor series, a Padé approximation or perhaps even a minimax rational approximation. The final step is to reconstruct sine(x) from N and sine(f).

We consider the first two steps in more detail to see how they may be designed for reliability and robustness. Our basic assumption is that the given argument x is exact. This is not often the case in practical situations, but a program as important as an elementary function routine should be designed to satisfy the most demanding user, and there are users with exact integer arguments. This emphasis on accuracy cannot be justified if the extra cost is excessive, but in most cases the more accurate routine costs only a few percent more to implement and use than the less accurate one.

Under the assumption of exact arguments the argument reduction step is critical in preserving accuracy. For the sine routine the expression

$$f = x - N*pi$$

must be evaluated carefully lest there be a loss of significance in f associated with taking the difference of two nearly equal quantities. The obvious step is to extend x to double working precision, a representation that is again exact, and calculate f in this higher precision. This is clearly too expensive to consider in most cases, but much the same effect can be achieved by careful reformulation of the calculation in the working precision. Let C1 and C2 be two constants whose sum represents pi to beyond the working precision, and let C1 be exactly representable in less than working precision. C1 = 201/64 works for most non-decimal machines. Then calculate

$$f = (x - N*C1) - N*C2 .$$

Now the loss of leading significant digits in the first term is compensated by a gain in trailing significant digits in the second term, provided the product N*C1 can be exactly represented in the registers, i.e., provided N, hence x, is not too large. The added expense is one stored constant and two operations.

The restriction on the size of x or N appears to be an added limitation on the domain of the function routine, but this is not really so. As x becomes large in magnitude the machine representation of x approaches an integer multiple of the machine representation of pi and there is little significance in f, hence little significance in the computed sine. A robust routine should warn the user of this situation rather than provide a random number for a function value. The limitation on the size of N merely clarifies the situation and establishes a reasonable boundary for the largest acceptable argument. Near this boundary point the function has already become "grainy", i.e., arguments which differ by only one unit in the least significant bit position return function values which agree to only a few leading significant bits.

Argument reduction steps for other functions differ in detail but follow the same general pattern. In almost every case careful handling of critical computations involving a mathematical constant will pay big dividends in accuracy with minimal expense. This assumes that the argument is exact, because there is no way to compensate for an unknown error in the argument. Table I shows the difference careful argument reduction can make for selected function computations on an H.P. 65 hand calculator. This machine is an older programmable 10-digit decimal machine. In the cases cited up to half of the precision has been lost in the usual argument reduction which is hardware on this machine, but all figures are correct when carefully programmed argument reduction is used (later H.P. calculators also return full precision results). The similar loss of half of the precision in a nominal 6-digit microcomputer can be serious.

Table I

Effect of careful argument reduction in selected cases on an H.P. 65 hand calculator

| Function | Sin(22) | Tan(11) | Ln(1.0001) |
|---|---|---|---|
| "True" Value | -8.8513 09290 E-3 | -225.95084 65 | 9.9999 50000 E-6 |
| "Hardware" Value | -8.8513 06326 E-3 | -225.95092 46 | 9.9999 00000 E-6 |
| Careful Argument Reduction | -8.8513 09290 E-3 | -225.95084 64 | 9.9999 50000 E-6 |

The argument reduction step is often the best place for two related programs to be merged. The sine routine can serve double duty, for example, by being used for the computation of the cosine as well. Because

cosine(x) = sine(x+pi/2)

we could calculate the cosine by adding pi/2 to the argument and then calculating the sine. But we must be careful! Careless addition of pi/2 will negate an accurate argument reduction. We can retain full significance in f if instead of adjusting x we adjust the value of N for the computation of f.

Details are to be found in [7] if they are not obvious.

The second step allows the programmer a great deal of freedom within limitations imposed by accuracy. In our sine routine, as in most routines, the most useful approximations come from the family of rational functions, a typical member being

$$R_{mn}(f) = P_m(f) \,/\, Q_n(f)$$

where $P_m$ and $Q_n$ are polynomials of degree m and n, respectively. The special case n=0 corresponds to pure polynomial forms such as truncated Taylor series. In true rational forms, those with both m and n nonzero, one coefficient may be chosen arbitrarily to satisfy additional numerical constraints, all other coefficients being scaled accordingly. Such scaling is often used to avoid loss of significance associated with wobbling precision on hexadecimal machines, or to save one multiplication in the evaluation of

$$Q_n(f) = ((q_n * f + q_{n-1}) * f + q_{n-2}) * f + \ldots$$

by setting $q_n = 1.0$. True rationals may also be rewritten as truncated continued fractions to avoid roundoff in the usual representation, or broken into a constant plus a rational correction term to better represent a slowly varying function. These and other possible maneuvers make true rational forms generally superior to polynomials for achieving reliability.

Approximations are most efficient when they preserve basic analytic properties of the function being approximated. In our example, sine(f) is an odd function, i.e., sine(-f) = -sine(f). This property is preserved by rationals of the form $f*R(f^2)$, and most sine programs use such an approximation. But programs which use the same approximation for all values of f may not be robust, because the intermediate quantity $f^2$ can underflow for $|f|$ sufficiently small. If $f^2$ is replaced by zero when underflow occurs, the underflow is probably non-destructive, but the resulting arithmetic interrupt and error message are still annoying. A robust sine program avoids the underflow entirely by trapping out |f| less than some threshold, say 2**(-t/2) where there are t bits in the significand of a floating-point number, and returning f for the function value. Again the added expense for a clean program is minimal.

The chip mentioned in the previous section undoubtedly portends the next major modification to machine hardware for scientific computation. This particular chip is intended for use with microcomputers, but there are clear indications that similar hardware or microprogrammed implementations of the elementary functions are also under consideration for maxicomputers [19]. Comparing the complexity of these functions with the complexity of floating-point arithmetic, and considering the lack of clean arithmetic hardware, there is genuine cause for concern about the quality of functions implemented in this way.

Calculator chips are our primary examples of this type of technology. Most of these use a CORDIC scheme [21] in the second step of the general computation outlined above. This involves a continued product expansion of the function instead of the usual rational approximation. The CORDIC method is fast and accurate in most cases, the major exception being the accuracy of logarithms for arguments close to 1.0. Chips in recent Hewlett-Packard calculators are even accurate in this case, but the details of the scheme used, presumably a modified CORDIC, are secret. Outside of the primary range the accuracy of the function still depends upon the argument reduction, a fact most chip designers ignore. Only a few calculators, again including recent Hewlett-Packard models, are as careful in argument reduction as we have advocated.

Verifying the reliability and robustness of a function program is as delicate a task as writing the program in the first place. Consider the simple problem of accuracy testing. The objective is to measure the error in computed function values given exact arguments. The problem is to devise a testing procedure which can do this without introducing additional error, systematic or random. The best testing procedures involve direct comparisons against higher precision computations, but such techniques are complicated and often difficult to implement, especially when the higher precision arithmetic must first be programmed. A second class of testing procedures measures the error in selected mathematical identities. These procedures are less discerning than the first ones, but are easily implemented. Carefully done, test programs based on identities distinguish between good and bad function programs and provide error statistics only slightly inferior to those provided by tests of the first kind.

As an example, one procedure for testing the sine function measures the relative error

$$E = [\text{sine}(x) - \text{sine}(x/3)\ (3 - 4\text{sine}^2(x/3))]\ /\ \text{sine}(x)$$

in the triple angle formula. Arguments are drawn randomly from intervals selected to minimize the subtraction error in E and to focus the accuracy test on one particular aspect of the sine routine. Arguments from the interval (0,pi/2) are used to test the routine when no argument reduction is needed, i.e., to test the basic computation of sine(f) for the reduced argument f; and arguments from the interval (6pi,13pi/2) are used to test the accuracy of the argument reduction scheme. In both cases, unless x and x/3 are exact machine numbers, however, test results are contaminated to the point where they are useless. There are simple ways for assuring that these exactness conditions hold, but we do not want to get into detailed numerical analysis here. We urge the interested reader to consult [7] for more information on these and other useful tests for the sine function, including a complete Fortran test program.

Table II gives results from selected runs of that test program on several different computer systems. An examination of the error statistics indicates which systems have sine routines with good basic approximations, good argument reduction schemes, and good alternate entries for the cosine function. Tests were also run using Basic 1.1 on the Datapoint 2200, where a curious blunder was detected in the sine routine. Computations were reasonably good for $|x| < 2pi$, but the routine returned the value of sine(4x) for x > 2pi. This blunder was not detected by the identity tests described above, but was detected by a different test designed for that specific purpose.

Table II

Results of random argument accuracy tests of sine and cosine routines based on triple angle formulas. All tests were 2000 uniformly distributed arguments in each interval. MRE is MAX(ABS(E)), the maximum absolute value of the relative error E, and RMS is the root-mean-square value of the relative error E.

| Machine, Library, and Fl. Pt. Significance | | Test and Interval sine(x) (0,pi/2) | sine(x) (6pi,13pi/2) | cosine(x) (7pi,15pi/2) |
|---|---|---|---|---|
| IBM 370/195 Extended Fortran 6 Hex (24 bits) | MRE | 16**(-4.84) | 16**(-4.84) | 16**(-4.82) |
| | RMS | 16**(-5.30) | 16**(-5.31) | 16**(-5.31) |
| PDP 11/45 Fortran DOS 8.02 (24 bits) | MRE | 2**(-22.06) | 2**(-22.26) | 2**(-11.37) |
| | RMS | 2**(-23.90) | 2**(-23.91) | 2**(-16.34) |
| Varian 72 Fortran E3 (22 bits) | MRE | 2**(-20.13) | 2**( -8.46) | 2**( -9.31) |
| | RMS | 2**(-22.20) | 2**(-13.45) | 2**(-14.69) |

As we indicated earlier it is not often that the data used is precise enough to justify our approach to accuracy. The real benefits are intangible. Kuki states that the major benefit is psychological [15]. The average user places great faith in the basic library programs, just as he trusts the arithmetic operations. He usually looks elsewhere for an explanation when computational error is unacceptably large. Programs written as carefully as we suggest can be demonstrated to be accurate, building the user's confidence. Less carefully written programs can be demonstrated to be inaccurate on exact data, destroying the user's confidence, even though they may be perfectly acceptable for processing his data. The penalty with the less accurate routine is that the user may be misled in his search for errors in his computation. In extreme cases he may even provide his own, often inferior, replacements.

## 5. Other Common Numerical Software

It is much easier to write good elementary function programs than it is to write good numerical software for other purposes. Elementary functions are simple mappings of one one-dimensional subset of the real numbers into another. Usually there are only a few paths through an elementary function routine, and it is the details of implementation, not the choice of algorithm, that decide how reliable and robust the routine will be. By contrast, most other mathematical processes of interest, even those representable in simple mathematical terms, are complicated mappings involving subsets of n-space. There are many paths through a typical subroutine for such a process. In this case it is primarily the choice of algorithm that determines reliability and robustness of software and not so much the details of implementation, although the latter cannot be ignored completely.

We do not expect microcomputers to be called upon very soon to do matrix eigenanalysis or solve stiff differential equations, but they probably will be expected to do such simple tasks as summing data, performing linear regression, calculating the Euclidean norm of a vector, and solving quadratic equations. In each case the obvious algorithm can give reasonable looking yet incorrect results without warning. The following example is adapted from Kahan [14].

Linear regression is the process of fitting a straight line to data in the least squares sense. A typical regression program in a microcomputer might be expected to automatically receive data from an online experiment until the end of the experiment is signalled, and then return the regression coefficients and perhaps the mean and standard error for the data. The classical equations are

$$Y = Mx + B$$

$$M = \frac{\Sigma xy - \Sigma x \Sigma y/n}{\Sigma x^2 - (\Sigma x)^2/n}$$

$$B = (\Sigma y - M\Sigma x)/n$$

$$\overline{x} = \Sigma x/n$$

and

$$s_x^2 = [\Sigma x^2 - (\Sigma x)^2/n]/(n-1)$$

where n is the number of data points (x,y), $\overline{x}$ is the mean, $s_x$ is the standard deviation, and Y is the value predicted by the linear regression [8]. A number of chips now used in hand calculators facilitate the use of these equations by providing one instruction which automatically augments each of the sums when given a new (x,y) data pair. There will naturally be a tendency to use these classical equations should such chips become available on microcomputers. Yet these equations can lead to surprising problems. For example, consider an implementation in single precision arithmetic on the IBM 370/195.

This machine uses hexadecimal floating-point arithmetic with a 24 bit significand, i.e., it nominally represents numbers to at least 7 significant decimal places. However, given the two points (1365.0, 1.0) and (1366.0, 2.0), the program returns the grossly incorrect values

| | Calculated | True |
|---|---|---|
| M | 0.5 | 1.0 |
| B | -681.25 | -1364.0 |
| $\overline{x}$ | 1365.5 | 1365.5 |
| $s_x$ | 1.0 | $1/\sqrt{2}$ |

without any indication of a malfunction. The problem in this particular case arises because $\Sigma x^2/n$ cannot be represented exactly in the machine, even though $\Sigma x^2$ can.

A slight modification of the computation will give correct results for this data, i.e., we can simplify the expression for M by multiplying the numerator and denominator by n, but that is not the important point. We are treating the symptom and not the disease when we do that. The new formulation will inevitably cause trouble with slightly different data. Adding the point (1367.0,.3.0) results in an error exit for division by zero, for instance, because now the two terms in the denominator agree to machine precision. The real difficulty is that the classical equations are inherently unreliable for numerical computation on a short wordlength machine. We must seek a different algorithm.

The equations for M and $s_x^2$ can be rewritten as

$$M = \frac{\Sigma(x-\overline{x})(y-\overline{y})}{\Sigma(x-\overline{x})^2}$$

$$s_x^2 = \Sigma(x-\overline{x})^2/(n-1)$$

which scales things so that none of the intermediate results becomes too large in comparison to the wordlength. Of course, this does not help much if the computation of $\overline{x}$ and $\overline{y}$ now requires that all data be kept in storage until the value of n is known. Fortunately that is not the case; running estimates of $\overline{x}_k = \sum_i^k x_i/k$ and of $a_k = \sum_i^k (x_i - \overline{x}_k)^2$ can be calculated by recurrence methods as follows [8,14]. Let $\overline{x}_1 = x_1$ and $a_1 = 0$. Then

$$\overline{x}_{k+1} = \overline{x}_k + (x_{k+1} - \overline{x}_k)/(k+1)$$

and

$$a_{k+1} = a_k + k(k+1)[(x_{k+1} - \overline{x}_k)/(k+1)]^2 .$$

This approach allows the computation of linear regression coefficients essentially to within roundoff in the coefficients regardless of the number of terms. Further, the regression coefficients, means and standard deviations can be obtained at any intermediate point without disturbing subsequent computations.

There are similar algorithmic problems associated with each of the other mathematical processes mentioned earlier [1,9,12]. All have been solved, and experienced numerical analysts are aware of the solutions. But the computations appear to be so trivial mathematically that computer users are constantly writing their own software for these tasks completely unaware that there are problems.

It is somehow ironic that the average user providing his own software is often no worse off than his colleague who relies naively on the software provided for him in a library. There appears to be no remedy for the situation. But microcomputer users are not alone; we are all victims of ignored history. The real challenge in numerical software was known to Hegel and Santayana.

## References

[1] J. L. Blue, "A portable Fortran program to find the Euclidean norm of a vector," to appear Trans. Math. Software.

[2] N. W. Clark, W. J. Cody and H. Kuki, "Self-contained power routines," Mathematical Software, J. Rice, ed., Academic Press, New York, 1971, pp. 399-415.

[3] W. J. Cody, "The influence of machine design on numerical algorithms," 1967 Spring Joint Computer Conference, AFIPS Conf. Proc., Vol. 30, Thompson Book Co., Washington, D.C., 1967, pp. 305-309.

[4] ------, "Software for the elementary functions," Mathematical Software, J. Rice, ed., Academic Press, New York, 1971, pp. 171-186.

[5] ------, "The construction of numerical subroutine libraries," SIAM Review 16, 1974, pp. 36-46.

[6] ------, "An overview of software development for special functions," Lecture Notes in Mathematics 506, Numerical Analysis Dundee 1975, G. A. Watson, ed., Springer Verlag, Berlin, 1976, pp. 38-48.

[7] ------, A Software Manual for the Elementary Functions, in preparation.

[8] A. I. Forsythe, T. A. Keenan, E. I. Organick, and W. Stenberg, Computer Science, A First Course, Second Edition, Wiley, New York, 1975.

[9] G. E. Forsythe, "What is a satisfactory quadratic equation solver?," Constructive Aspects of the Fundamental Theorem of Algebra, B. Dejon and P. Henrici, eds., Wiley-Interscience, New York, 1969, pp. 53-61.

[10] M. Ginsberg, "Numerical influences on the design of floating-point arithmetic for microcomputers," this proceedings.

[11] IBM System/360 Scientific Subroutine Package, Version III Programmer's Manual, Fifth Edition, IBM Systems Reference Library, GH20-0205-4, IBM, White Plains, New York, 1970.

[12] W. Kahan, "Further remarks on floating-point summation," Comm. ACM 8, 1965, p. 40.

[13] ------, "Implementation of algorithms, part I," Tech. Report 20, Dept. of Comp. Sc., University of California, Berkeley, 1973.

[14] ------ and B. N. Parlett, "Can you count on your calculator"," unpublished mss., 1977. (A mutilated summary appears in Electronics, November 25, 1976, pp. 77-78.)

[15] H. Kuki, "Mathematical function subprograms for basic system libraries - objectives, constraints, and trade-offs," Mathematical Software, J. Rice, ed., Academic Press, New York, 1971, pp. 187-199.

[16] ------ and W. J. Cody, "A statistical study of the accuracy of floating point number systems," Comm. ACM 16, 1973, pp. 223-230.

[17] M. D. Maples, "Floating point package for INTEL 8008 and 8080 microprocessors," Report UCRL-51940, Lawrence Livermore Laboratory, Livermore, California, 1975.

[18] MM57109 MOS/LSI Number-Oriented Microprocessor, Product Announcement, National Semiconductor Corp., Santa Clara, California, March, 1977.

[19] G. Paul and M. W. Wilson, "Should the elementary function library be incorporated into computer sets," Trans. Math. Software 2, 1976, pp. 132-142.

[20] B. T. Smith, J. M. Boyle, and W. J. Cody, "The NATS approach to quality software," Software for Numerical Mathematics, D. J. Evans, ed., Academic Press, New York, 1974, pp. 393-405.

[21] J. S. Walther, "A unified algorithm for elementary functions," 1971 Spring Joint Computer Conference, AFIPS Conf. Proc., Vol. 38, AFIPS Press, Montvale, N. J., 1971, pp. 379-385.

# Numerical Influences on the Design of Floating-Point Arithmetic for Microcomputers

by

Myron Ginsberg
Department of Computer Science
School of Engineering and Applied Science
Southern Methodist University
Dallas, Texas 75275

Abstract

The increasingly sophisticated applications for microcomputer systems require an efficient and reliable general-purpose floating-point facility. This paper focuses attention on numerical behaviorial problems that should be considered both when designing and using floating-point arithmetic on microprocessors. Topics discussed include tradeoffs amongst floating-point representations, deviations of behavior from that exhibited by the real number system, numerical algorithmic difficulties, and testing techniques for assessing the performance of floating-point computations. A comprehensive bibliography is also included which serves as a guide to the literature on floating-point arithmetic and its influences on mathematical software development.

## 1. Introduction

The increasingly sophisticated applications for present and future microprocessor systems require an efficient and reliable floating-point facility with an extensive support library of practical functions that are easily accessible to users; work in this area is still in an embryonic state of development and could therefore benefit from the floating-point experience acquired from efforts on larger machines. This paper focuses attention on some of the numerical problems involved in selecting and using floating-point arithmetic on microcomputers. It is hoped that knowledge of floating-point behavior traits can benefit the microprocessor community by helping its members to understand and anticipate problems in designing and using general-purpose computer arithmetic with future systems.

The creation, utilization, and testing of a floating-point capability involves many factors with a myriad of interactions. Throughout this paper we shall indicate numerous behaviorial features and how to detect and cope with them. In Section 2, floating-point representations are examined along with tradeoffs amongst base 2, 8, and 16 arithmetic. The next section deals with discrepancies between floating-point behavior and that exhibited by the real number system. The penultimate section is devoted to numerical difficulties associated with certain algorithmic features. Testing techniques to assess the reliability of a computer's floating-point facility are discussed in the final section. The interested reader seeking further information concerning the subject matter of this paper is encouraged to examine the comprehensive bibliography in the appendix; it contains references to works on the behavior of various floating-point arithmetic systems and other computational environmental influences affecting the development, testing, and use of reliable mathematical software.

## 2. Examples of Floating-Point Representations

The internal floating-point representation in most computer systems consists of the components shown in Figure 1; in large machines, the floating-point format is contained in a single word whereas several words are usually needed for a mini- or microcomputer due to the shorter word length. S is the sign bit and indicates whether the number is positive or negative. The mantissa is composed of a T-digit fraction expressed in a number base, $\beta$, called the radix, which is selected for the internal floating-point design; the radix point is usually assumed to be to the immediate left of the first β-ary digit, $d_1$, of the mantissa. The exponent segment value, denoted by e, is either a signed number or a biased representation, i.e., a fixed constant is added to the exponent value to eliminate the need for an explicit sign bit. The base β is raised to the power indicated by the exponent. The internal form of a floating-point number, X, is expressed mathematically as $X = \pm \cdot d_1 d_2 \ldots . d_T \times \beta^e$ where each β-ary digit, $d_i$, is an integer between 0 and β-1, inclusive. X can also be defined external to the machine as a base 10 number $X = \pm \cdot C_1 C_2 \ldots C_{T'} \times 10^f$ where f is an appropriate exponent of 10 and each $C_i$ is between 0 and 9, inclusive; T does not necessarily equal T′. A number exactly representable in base 10 need not be in another base and vice versa; for example, $\frac{1}{10}$ is represented exactly by .1 in base 10 but has no exact equivalent in base 2. Thus error can be introduced by computer conversion procedures to input or output floating-point numbers. Most computers use bases 2, 8, or 16 in their floating-point models. Figure 2 lists some attributes of typical floating-point designs on existing machines.

Operands and resultants of the basic floating-point arithmetic operations are usually expressed in "normalized" form, i.e., the mantissa is specified as a fraction with a nonzero first β-ary digit, $d_1$. An unnormalized number, $(d_1 = 0)$, can be normalized by left shifting the base β mantissa digits until $d_1 \neq 0$ and appropriately decreasing the exponent value. When normalizing, it is helpful to have available one or more "guard" digits so that as the shifting takes place, significant nonzero β-ary digits enter into the right end of the mantissa rather than non-significant zeros; such guard digits participate in an intermediate stage of the arithmetic operations within the machine's registers and are also used for rounding purposes. Normalization is performed with respect to β-ary digits, $d_i$; thus a normalized base 2 number must have a leading nonzero mantissa bit whereas a normalized octal or hexadecimal number can have up to two or three leading zero bits, respectively. This variation in the number of leading zero bits present in high base models contributes to fluctuations in numerical performance.

The following factors should be considered when designing and using a floating-point facility: (1) accuracy of basic arithmetic operations and all the library routines; (2) number base used for internal format; (3) total number of bits in the floating-point representation; (4) mantissa versus exponent lengths; (5) accuracy of input and output conversions; (6) hardware tradeoffs, such as number base versus floating-point range or speed versus size of floating-point model; (7) appropriate range of representable numbers for microcomputer applications; (8) rounding procedures, such as biased or unbiased rounding versus truncation. Clearly with so many interdependent influences it is difficult to find a single, ideal format. Both designers and users will have to collectively assess their priorities in order to narrow down the choices of candidates for the best arithmetic system to meet their needs.

To illustrate some of the interactions amongst the above-mentioned factors, we will examine a few simple cases. Let us consider 8-bit mantissas in hexadecimal and binary as well as 6-bit mantissas in octal and binary. For all cases we will assume normalized representations and exponent values of 0, 1, or 2. The vertical lines in Figures 3, 4, 5, 6, and 7 depict the exactly representable numbers for each base and format in the range 0 through 5, inclusive. The corresponding negative regions are mirror images of those positive regions shown in the aforementioned figures. In all the examples, the same basic fractional unit ( $\frac{1}{256}$ ) is used to measure the distance between adjacent representable numbers.

Several interesting relationships can be observed in these four models. Although we are dealing with finite representations of the real (infinite) number system and therefore must expect gaps between successive representable numbers, it does seem somewhat surprising that the spacings are not equal, i.e., the interval between any two consecutive representable numbers varies in size depending on its location and becomes larger as the distance from 0 increases. This situation is also observed when moving away from zero in a negative direction. For example, the spacings between the floating-point representation of 1 (or -1) and its immediate predecessor and successor are $\frac{1}{256}$ and $\frac{1}{16}$, $\frac{1}{256}$ and $\frac{1}{128}$, $\frac{1}{64}$ and $\frac{1}{8}$, $\frac{1}{64}$ and $\frac{1}{32}$ for bases 16, 2(8 bits), 8, and 2(6 bits), respectively. Only between successive powers of $\beta$ are the gaps equal between consecutive representations.

The non-uniform distribution of the representable numbers contributes to the violation of several properties of the real line (see Figure 10). For example, 16 and $\frac{1}{2}$ are exactly representable in our base 16 format but their sum is not and would have to be assigned a value of 16 or 17 in this case (nearest representable numbers to result). Similarly, it can be shown that subtraction, multiplication, and division with exactly representable numbers do not necessarily produce exact results. In these circumstances, hardware or software procedures have to be used to select the best representable number to approximate the exact results; such decisions are usually made with the aid of guard digits which participate in a rounding scheme. Other discrepancies from the behavior of the real number system are discussed in the next section.

We observe that our larger number base models (8 and 16) have more members in the interval [-1, 1] than do the corresponding base 2 models but fewer members than their binary counterparts as we move further away from zero; this can be verified by the interval representation counts given in Figures 8 and 9. Also in both the 8-bit and 6-bit cases, the base 2 formats exhausted their ranges of representable numbers before the higher base systems did. It should be noted, however, that within the base 2 range, (-4,4), the number of floating-point representations in the hexadecimal and octal models is 75% and 83%, respectively, of the corresponding number of base 2 representations. Thus higher number bases provide wider ranges for the same number of bits than their binary counterparts but offer fewer exact representations outside the interval [-1, 1].

For a microprocessor system we would most likely select a floating-point model with multiples of 8-bit bytes for exponent and mantissa components. One might at first consider choosing a hexadecimal format with say three 8-bit parcels for the mantissa and one 8-bit segment for sign and exponent along with chopped base 16 arithmetic; this scheme provides a wide range of approximately $10^{\pm 75}$ and tends to reduce alignment and normalization shifting (see Reference I-18). But such a model would have the equivalent of only approximately 7 decimal digits and would suffer from the same representation problem exhibited by our base 16 example, i.e., fewer exactly representable numbers than a base 2 format in regions outside [-1, 1]. Furthermore, the adoption of such a model would probably be unwise in view of a history of numerous complaints from scientific users concerning the numerical behavior of such hexadecimal machines.

Selection of a base 16 system per se is not bad; however, existing implementations have tended to produce a combination of unfortunate effects that have led to aberrant arithmetic behavior which could have been avoided or minimized by some modifications. For those microprocessor designers who believe that the hardware implementation advantages of base 16 arithmetic justify the adoption of a hexadecimal system, it is strongly suggested that alterations to previous designs should include the use of a longer mantissa, say 5 or 6 bytes (would provide the equivalent of approximately 11 to 14 decimal digits, respectively),two guard digits, and an unbiased rounding scheme (see References I-4, I-9). Several computational studies (see References I-1, I-4, I-9, I-37) seem to indicate that the combination of base 16, short mantissa size,and truncated arithmetic should definitely be avoided.

To promote a high level of numerical reliability for scientific computations performed on microcomputers, it seems advisable to provide a floating-point facility with more than the equivalent of 6 or 7 decimal digits and at least comparable to pocket calculators using 10 to 13 decimal digits. Oftentimes scientific calculations involve the use of one or more library routines (each of which introduces varying amounts of error depending on argument values) along with program loops in which the total floating-point error propagation can easily reach levels that seriously erode the effectiveness of 6 or 7 decimal digits. Such situations could jeopardize applications which require uniformly reliable results. To produce a final output with only a few accurate leading digits may require substantially more digits at intermediate stages of the computation. Further accuracy enhancement of a floating-point facility can be achieved by selecting an appropriate rounding scheme; several computational studies (see References I-1, I-4, I-9) seems to suggest the advantage of using rounding (with two guard digits) rather than truncation, particularly unbiased rounding (see References I-4, I-9) referred to as R* mode.

On the basis of previous experience on large machines, it might be prudent to select a base 2 representation with a large mantissa, say 4 or 5 bytes (would provide the equivalent of approximately 10 to 12 decimal digits, respectively), R* mode, and 2 guard digits. As a matter of fact, a simulation study (see Reference I-1) indicates that if, in addition, we use an implicit representation in our binary model (i.e., the leading mantissa bit is implicit with the restriction that only normalized representations of nonzero numbers are permitted in the system) then such a floating-point system "is roughly equivalent to carrying one more decimal place" than a base 16 representation with truncated arithmetic over essentially the same range. The adoption of the binary model while increasing accuracy might have the net effect of a decreased range and possibly slower execution than a base 16 counterpart; however, the former could be alleviated by use of a longer exponent field and the latter might be remedied by hardware or architectural changes (or the use of a faster machine to perform the arithmetic) to improve the floating-point performance rate when working with multi-byte mantissas. Another alternative is to provide a relatively fast single precision, floating-point capability and a slower double precision option which provides both an increase in exponent range and mantissa size as is done on the Univac 1108; mixed mode operations would have to be handled carefully to avoid any unnecessary arithmetic anomalies.

The final form of the floating-point facility will be strongly influenced by hardware speed, software support, and the restrictions imposed by microprocessor applications. The selection of a "clean" floating-point model is not sufficient to insure trouble-free numerical behavior. The hardware and software (especially the library routines) used to maintain the floating-point system are capable of introducing their own arithmetic peculiarities and/or intensifying finite representational problems to produce further deviations from the behavior exhibited by the real number system. Some of these difficulties are discussed in the next section.

## 3. Floating-Point Behavior Problems

There are many anomalies between floating-point behavior and that exhibited by the real number system; Figure 10 summarizes several of these situations and Reference I-17 even defines specific floating-point representations for which a property of the real number system can fail. Most of the deviations from real line behavior presumably affect a minority of floating-point numbers in a particular system; just how many of these "fringe" cases are important for a particular application is dependent on the number base, mantissa size, and rounding rules enforced by the floating-point facility. It is probably safe to assume that one or more of these conditions will occur even in small to moderate size scientific computations.

The extent of deviant behavior is not only depéndent on the attributes of the floating-point representation but also on the influence of high-level language compilers. This computational aspect should be of concern to microprocessor users because of the increasing availability of high-level languages including BASIC, FORTRAN, subsets of PL/I, and ALGOL. A numerical result produced with object code from one compiler may change when used with code from another compiler or from another level or option with the same compiler. This happens because there is usually a 1 - to - many relationship between the user's high-level language program and possible machine language versions. Oftentimes, particularly with optimizing compilers, the sequence of floating-point arithmetic operations is altered from that specified by the programmer in order to produce object code which will execute rapidly; unfortunately, many times the re-arrangement of operations assumes that all the rules of the real number system are valid when applied to floating-point numbers. As indicated from the comments above, this assumption is not necessarily true. A cautious user aware of this difficulty may insert many sets of explicit parentheses to prevent arithmetic sequence re-orderings by the compiler; however, some compilers are so "clever" that even this precaution does not preclude situations in which the compiler still hinders programmer attempts to avoid numerical difficulties as in the case to try to avert subtractive cancellation reported by Cody (see Reference II-3). Another situation which was constructed by Kahan (see Figure 11 and either Reference II-15 or IV-6) illustrates that some compilers perform arithmetic at translation time with explicit constant expressions using a different sequence of round-off rules than those employed at execution time; thus two programs which appear to be equivalent, except that one uses explicit constants in an arithmetic expression and the other uses variables which have been previously set equal to the corresponding constants, may not produce the same numerical results. Even in cases where compilers cause no numerical difficulty they can introduce significant variation of execution times amongst object codes for very similar source programs; for example, Parlett and Wang (see Reference I-29) present some cases occurring in linear equation solvers where there is as much as a 50% to 90% variation in execution times amongst object codes.

Input and output floating-point conversion procedures are another source of floating-point behavior problems since numbers exactly representable in one number base are not necessarily exactly representable in another base. Matula (see Reference I-11) has developed some results to determine what should be the minimum size of a computer's floating-point mantissa so that a number input to the machine could be output as exactly the same number or deviate from the original number by one unit in the last place. Notice from the table in Figure 12 that to maintain the same number of decimal digits for this relationship to hold, a hexadecimal machine usually requires more bits in its internal representation than an octal machine and an octal machine always requires more bits than a binary computer. Furthermore, we observe that one implication of this relationship is that the number of decimal digits serving as input data to a program and which also minimizes conversion error will vary considerably across existing machines; thus, for example, with a 5 or 6 byte binary mantissa we could input approximately 12 to 14 decimal digits whereas with a 3 or

4 byte hexadecimal mantissa we could input only 6 to 8 decimal digits and still satisfy the above-mentioned assumption.

Some microprocessor designers may believe that an easy way out of the floating-point behavior dilemma is to perform arithmetic using pocket calculator chips (see Reference I-55). Even assuming the speed differential, interface, and display problems can easily be resolved, this can still be an unsatisfactory alternative at the present time. The main reason is that most pocket calculators introduce their own serious anomalies (see References I-50, I-51). At least one manufacturer uses different sizes for the accumulator, stack elements, and memory registers in such a manner that the order of the operands can affect the results of a single arithmetic operation, i.e., the commutative law of addition (A+B = B+A) or multiplication (A*B = B*A) can be violated on such calculators. Peculiar or inconsistent use of guard digits and a penchant for truncation rather than rounding cause further difficulty. Additional behavior problems on many calculators are introduced by the non-uniform error behavior or poor quality of the elementary functions over the range of representable arguments. Thus at present, with a few exceptions (such as the clean arithmetic and elementary functions on the HP67 or 97), it would probably be unwise to utilize most of the currently available pocket calculator chips in a microprocessor system if consistent and highly reliable numerical results have top priority.

## 4. Algorithmic Implementation Considerations

From the above discussions it should be apparent that to produce highly reliable mathematical applications software the algorithm selection and implementation phases cannot afford to be oblivious to the computational environment. Depending on the degree of programmer awareness and expertise, it is possible to create several implementations each of which can produce different numerical results for the same problem on the same machine with the same compiler; of course, additional variations can often be introduced by utilizing several translators or by choosing different options with the same one. When selecting an algorithm the user should attempt to determine its limitations for solving his problem and how the calculations can best be implemented in a particular environment to maintain numerical integrity. This means we must be concerned with such matters as: choosing appropriate termination criteria and error tolerances; attempting to minimize round-off errror propagation; determining how, when, and where additional precision is needed; monitoring error and/or other factors to inform users of the reliability of the computed result. In this section and the next we describe some actions for dealing with these issues.

Scientific applications programmers should consider employing routines to automatically determine computational environment parameters which, in turn, could be utilized in implementing algorithms and in promoting program transportability to other machines. A simple example of such a routine is that written by Malcolm (see Reference V-12) and modified by Gentleman and Marovich (see Reference V- 9); this code determines the floating-point number base, mantissa size, and whether chopping or rounding is used. By inserting within an algorithm's implementation this or other such program segments to determine the computational setting then such items as error tolerance, scaling factors, generation of appropriate number of digits for program constants, termination criteria, etc.

could be specified in terms of the environmental features. The quality of scientific applications software could be greatly enhanced by use of arithmetic, algorithmic, and systems parameters; with such information, a program designer could provide his own fixups for overflow or underflow conditions and other situations where the software might fail due to the computational surroundings. This could considerably ease the burden of adapting programs to new machines. A discussion of what factors should be considered as basic machine parameters is given by Cody (see Reference V-5).

Additional issues regarding the production of highly reliable and transportable math software are being considered by IMSL (International Mathematical and Statistical Libraries, Inc.), NAG (Numerical Algorithms Group), NATS Project (National Activity to Test Software), and the PORT Library Project. Microprocessor applications programmers could benefit from a perusal of the activities in this area. Section V of the bibliography in the appendix to this paper lists several references. Also Reference II-11 provides some examples of portable codes for several problem areas in numerical analysis.

Because of the eccentricities of floating-point arithmetic, certain precautions should be taken when implementing an algorithm. For example, when summing floating-point numbers of the same sign, round-off error can be reduced by summing in order from the smallest in magnitude to the largest; this can be particularly helpful in inner product calculations as well as in summing terms of certain convergent series and can provide noticeable improvement over the worst case summation using the reverse order of terms. For more general cases involving the summation of both positive and negative terms, various techniques have been devised; some of these are discussed in References II-12, II-14, II-20, II-26, and II-34.

Another very common situation needing attention involves a phenomenon called subtractive cancellation, i.e., where two numbers of approximately the same magnitude but opposite signs are subtracted from one another; this is a potentially hazardous condition in that the relative error could be quite large even if the difference is small in which case the resulting error propagation could have serious repercussions on the accuracy of succeeding calculations. To avoid or reduce numerical deterioration from this effect, the algorithm can either be re-written to eliminate such situations or a higher precision can be invoked at some intermediate stage of the calculation before the subtractive cancellation occurs. Further examples of specific algorithmic computational difficulties are described in References II-9, II-15, and II-32.

The decision as to whether or not to use higher precision is problem, algorithm, and machine dependent; the user should be able to determine if he will obtain sufficiently more correct digits to justify the increase in time and/or memory requirements involved in using multiple precision. Let us examine a few possibilities. Figure 13 illustrates a situation in which the ratio of double precision to single precision time for the program segments used for inner product calculations in the solution of linear systems of equations is approximately 2.47 for the IBM 370, 6.31 for the CDC 6500, and 1.81 for the Univac 1106; note, however, that for all three machines the single precision version is never more than one digit less accurate than the double precision version. For the IBM machine, which has the equivalent of only 6 or 7 decimal digits in single precision, the increase in time to obtain an extra digit can probably be justified. For the Univac machine, which has the equivalent of 7 or 8 decimal digits in single precision, the relatively small increase in time could also be justified for selection of the double precision version. In contrast, for the CDC computer, which has the equivalent of 14 or 15 decimal digits in single precision, the substantial increase in time required for the double precision segment could probably not be justified in most cases, especially since only one more correct digit would be obtained. Of course, a final judgment in this example is not always clear cut; situations can arise when one additional correct digit is important. The point here is to illustrate some of the possible tradeoffs. Quality math software could provide the option for the user or the program to decide when to use more precision.

Another example demonstrates a compromise between single and double precision, namely partial double precision; intermediate results of arithmetic operations involving single precision operands are performed in double precision and the final results is truncated to single precision. In Figure 14 a simple differential equation (which can be solved exactly) is solved by a numerical technique. All the variables in the modified Euler formula are in single precision. The partial double precision version of this equation is obtained by replacing the expression for the product of the two terms on the right-hand side with DBLE (H) * DBLE $(F(X_K + \frac{H}{2}, Y_K + \frac{H}{2} * F(X_K,Y_K)))$. The net effect of this change is that the product is computed in double precision along with the following addition and then the resultant is chopped to single precision because $Y_{K+1}$ is a single precision variable. Computations on this problem were performed with decreasing step sizes on an IBM 360/67 (hexadecimal arithmetic with approximately 6 or 7 decimal digits) and a PDP-10 (binary arithmetic with approximately 8 decimal digits). We observe that for the single precision version, error decreases up to and including the N = 9 case but increases slightly on the IBM machine for N = 10. Note that the PDP-10 single precision results has a noticeably smaller error for N = 10 than does the IBM 360; however, for the partial double precision versions, the IBM results for N = 10 is significantly improved over its single precision counterpart but there is no such substantial improvement for the PDP-10. This example demonstrates that for a specific algorithmic implementation on a specific machine for a specific problem it may be advantageous to use partial double precision (IBM 360/67 in this case) whereas it may not be worthwhile on another machine (PDP-10 in this case). It would be very hazardous to generalize from this example since the aforementioned observations are very problem, algorithm, and machine dependent; however, this illustration does indicate some options the applications programmer should consider when designing his software.

One of the most insidious influences upon algorithmic implementations is the effect of the elementary library support functions such as sin, cos, log, etc. The microprocessor community is rapidly increasing its exposure to such functions as more and more high-level language facilities are becoming available. Most users automatically presume that the associated errors are negligible or at least uniform over all arguments. Such persons are strongly urged to carefully read References I-35, I-49, I-50, I-51, II-5, II-9, and II-15. Some pocket calculator users have already become victims of the oddities exhibited by some of these functions (see References I-50, I-51). Sometimes the numerical performance of a user-written program can be degraded by an elementary library routine which does not even explicitly appear in the program but is employed by another function which is explicitly present. (This situation also occurs on pocket calculators.) Another problem is that range reduction techniques employed by some of these functions produce very poor results.

The persons responsible for providing such library routines should thoroughly examine their behavior rather than passively accept those used on large machines and tacitly assume that they are satisfactory. Relatively little has been done to extensively check out the numerical reliability of library routines currently available as part of a commercial high-level language facility;

some of the implementations are very naive. The serious microprocessor user who intends to perform scientific computations should consider applying some of the available testing techniques (mentioned in Section 5) to check out the numerical behavior of his library routines. A few algorithms for some of the elementary functions as well as implementation considerations are given in References II-2, II-5, II-8, II-15, II-17, II-29, II-33.

The programmer should try to provide high quality, robust implementations of his algorithms; such software should warn users of potentially poor results, monitor or estimate error, and be thoroughly tested and well-documented, including clearly defined program limitations and a performance profile. A table like that given in Figure 15 would be helpful in indicating to the user how the error tends to vary as the argument range is changed. If more than one algorithm is used (usually called a polyalgorithm), the switching criteria should be examined to determine if the borderline cases are being handled properly. Furthermore, the program designer should make sure that his thoughtfulness in implementation is not sabotaged by the compiler; for example, Cody (see Reference II-3) cites a case in which to avoid subtractive cancellation, 1-Y was re-written as (.5-Y) + .5 but unfortunately the compiler treated this as 1-Y when generating code.

The moral is to know the behavior of the floating-point arithmetic, compiler, elementary functions as well as the algorithm and problem if you want to provide high quality mathematical software. As Figure 16 indicates, the sources of error in problem solving are all closely related; one false move at one stage can cause severe repercussions elsewhere. Cody discusses the attributes of quality software in Reference I-49. Section II in the bibliography lists sources of algorithm design experiences.

## 5. Testing Techniques for Evaluating Numerical Reliability

Several procedures are available for assessing the quality of a floating-point support facility and mathematical applications software. In this section we briefly describe some of the approaches and their limitations. Sections III and IV of the bibliography in the appendix give references to many of the techniques employed for error monitoring and performance testing.

A common practice has been to check out most routines for only a very small sampling of input data; usually a few cases with known results are examined along with some additional cases with random arguments. This approach, although widely used, is unsatisfactory in this form for adequately testing math software reliability because the use of a collection of pseudo-random arguments cannot insure that all numerically aberrant behavior for specific data values or subintervals of representable numbers will be discovered. This situation is further complicated by the influences of computerized floating-point behavior which is dependent on the number base selection, mantissa and exponent lengths as well as arithmetic round-off rules; thus, data sets which might adequately exercise the range of representable numbers of one computer might be entirely insufficient for another machine which possesses different floating-point attributes.

Modifications of the above-mentioned approach have been proposed to improve the situation. Cody (see Reference IV-5) has suggested that the entire range of representable numbers should be exercised for the input arguments of a tested routine by the use of both selected bit patterns and carefully-controlled sets of pseudo-random data; he advocates that the complete floating-point number range be decomposed into a set of sub-intervals and then a collection of random bit patterns be tested in each interval, and the computed output compared with exact results, entries from published tables, and/or numerical results from calculations performed in higher precision than is employed by the tested program.

If the amount of error associated with the computed results of the test routine varies substantially in any sub-interval, then that interval is further subdivided and testing commences with pseudo-random arguments in these new intervals in an attempt to isolate the argument ranges causing numerical problems. In addition, sub-interval boundary points, selected bit patterns based on the floating-point representation, and any cross-over and neighboring data points where the routine changes algorithms are also tested. From this procedure a table can be generated like the one shown in Figure 15 in which argument range sub-intervals are given along with the corresponding N-bit errors, maximum relative error, and the root-mean-square error; such a table can warn the potential user of argument ranges to avoid as well as indicate the overall program accuracy. This approach and variations of it (see References IV-16, IV-17) could be useful in testing library and applications routines for microprocessors.

Of course, like all other available techniques, it cannot guarantee to locate all inaccurate situations; only an exhaustive testing of all possible representable combinations of input data values would do that and this is usually considered to be too costly and/or impractical. The above-mentioned suggested modifications do, however, significantly improve the chances of detecting problem cases. At the present time there is no automatic generation of selected bit patterns, thus requiring that such data be hand-generated and changed from machine to machine because of computer arithmetic dependencies. Bit patterns are used rather than decimal floating-point numbers in order to separate the transmitted error induced by the conversion of base 10 input to internal floating-point format from error induced by exact input data. The effects of the conversion errors can be examined separately, if desired.

During both the phases of program design and production use, it is helpful of assess numerical reliability with the aid of error-measuring techniques.

The "textbook" type of error analysis is usually not too effective in practical situations; it tends to be concerned only with the truncation error (or often a non-optiomal bound on this error) associated with the creation of an algorithm and usually totally ignores computer round-off error propagation introduced by the implementation including error from use of the library support routines. Some typical error analysis approaches can be classified as follows: 1) error-bounding schemes; 2) forward error analysis; 3) backward error analysis; 4) multiple precision arithmetic. A compilation of references to these techniques is given in Section III of the bibliography in the appendix.

Error-bounding methods produce bounds to the computed solution. The most well-known of these schemes is interval analysis developed by Moore and associates (see Reference III-4). It is based upon use of a closed interval associated with each input variable and accompanying arithmetic rules for interval operations to trace through the calculations associated with a particular routine. The computed result of a problem in n-dimensions is an n-dimensional parallelpiped which contains the exact solution. For large computations, the widths of the intermediate intervals can grow rapidly and often require substantial computational overhead time. Although interval analysis can be time-consuming, it can be applied to a wide range of problems. Improvements and variations of interval analysis are given in Reference VI-1 and newer results are in Reference III-5.

A forward error analysis involves the calculation of a result and then a comparison of it to some reference value known to be exact and/or computed to higher precision than that used in the results being evaluated. Comparisons with published table entries and multiple precision output, such as is often done in argument testing, are examples of forward error analysis. Cody, Hammer, and others (see References IV-5, IV-9, IV-19) make use of mathematical identities or a Monte Carlo approach to perform a forward error analysis for parameter selection and evaluation.

Backward error analysis has been developed primarily by Wilkinson (see References I-47, I-48). With the application of this approach it is assumed that the computer results of a routine is the exact solution to some problem and an attempt is then made to determine how close the problem solved is to the original problem which was intended to be solved. The presumption is that if we are dealing with well-conditioned cases then if both problems are close then their results will also be close and appropriate bounds can be determined. This approach has also been utilized by Stoutemyer (see Reference I-44) with the aid of a symbolic manipulation system to generate symbolic floating-point round-off error expressions for each stage of the calculation and to use these expressions to aid in the analysis.

Work based on Wilkinson-type backward error analysis has also been used to develop an approach to automatic stability analysis (see References I-39, I-40). This technique is concerned with obtaining an algorithm's worst case round-off error rather than attempting to monitor error associated with each possible data set. The approach has great potential for economically determining if a method is numerically unstable. This is accomplished by the use of FORTRAN programs which define a heuristic search for a specific input data set that produces large round-off errors for the algorithm being tested. Since the computerized approach is heuristic, there is no guarantee that even if one or more such data sets exists, the program will necessarily find one. The method has been successfully applied to several numerical linear algebra methods and can be used in other problem areas with algorithms involving algebraic processes which satisfy certain limitations.

Multiple precision arithmetic is another alternative for assessing numerical reliability. The main premise is that increasing the precision under which a mathematical routine is executed can lead to extremely accurate results which, in turn, can be used to judge the accuracy of a program's single and double precision performance. Most multiple precision packages are defined in software and can therefore be very slow. It has been estimated (see Reference I-5) that execution time increases linearly with precision for addition and quadratically for multiplication; time increases can be far worse for extended precision function evaluations.

There are at least three packages known to this author which attempt to provide a portable extended precision facility. The system constructed at the National Bureau of Standards (see Reference III-12) is written in ANSI FORTRAN and can be used with a precompiler to scan certain super precision data types; it also includes a FORTRAN extended precision library of standard functions and permits multiple precision calculations in bases 2 to 16. The Hull and Hofbauer package (see Reference III-9) was originally written in ALGOL W and runs on the IBM 360/370 series; a FORTRAN version is currently under development. Brent (see Reference III-7) has constructed 97 routines written in ANSI FORTRAN; his package has been run on a PDP-10 and 11, IBM 360/370 series, Univac 1108 and 1110, and on a CDC Cyber 70, model 76. All three packages may be of interest to the microprocessor community both as a source of ideas as to how to design extended precision elementary functions and as a tool for checking out the numerical results of existing or proposed implementations of library and applications routines.

In addition to numerical reliability and error measures, some other commonly used performance criteria include central processor time, overhead time, number and type of arithmetic operations performed, number of function and derivative evaluations, storage requirements, cost, efficiency, and stability region. Definition of cost, efficiency, and reliability can vary considerably. References IV-11, IV-12, IV-15, IV-20 discuss various performance standards. Informal collections of test problems have evolved and have been used in several areas;

References IV-7, IV-12, IV-13, IV-15, IV-20 list some available test sets and others are given in Reference VI-3.

Numerous computational case studies have been performed. Analysis of their results should include an awareness of performance criteria, test case selection, computer algorithms and implementations investigated, and the computational environment in which the programs were executed. The methodologies employed in these studies should be of interest to microprocessor applications programmers. Some of the more extensive studies include those mentioned in References IV-2, IV-7, IV-12, IV-15, IV-20 and others listed in Reference VI-3.

It is hoped that from the comments, examples, and discussions in this section and in the rest of the paper, the reader has become aware of and appreciates the problems in designing, testing, and using a floating-point facility with its supporting hardware and software. In the next few years, the increasing use of microprocessors for such applications as financial calculations, sophisticated games, process control, and optimization should create a demand for high speed and reliable floating-point arithmetic. If the reader carefully considers the issues presented in this paper and in the references in the accompanying bibliography, he should be able to benefit from past experiences on large machines and be well on his way to intelligently designing and using a high-quality floating-point facility.

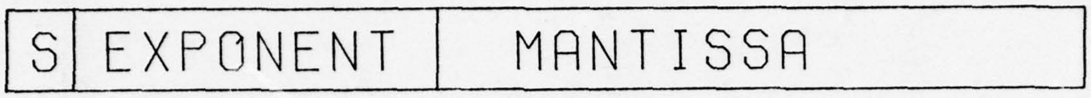


Figure 1: Format for Internal Floating-Point Representation

| | IBM 360 and 370 Systems | Burroughs B5000 Series | CDC 6000,7000,Cyber 70 | DEC PDP-10 |
|---|---|---|---|---|
| word length (bits) | 32 | 48 | 60 | 36 |
| $\beta$ | 16 | 8 | 2 | 2 |
| T | 6 | 13 | 48 | 27 |
| range of e in base $\beta$ | $-64 \le e \le 63$ | $-63 \le e \le 63$ | $-1022 \le e \le 1024$ | $-128 \le e \le 127$ |
| range of e in base 10 | $\simeq 10^{\pm 75}$ | $\simeq 10^{\pm 69}$ | $\simeq 10^{\pm 300}$ | $\simeq 10^{\pm 38}$ |
| exponent representation | biased | signed | biased | biased |
| exponent length (bits) excluding sign | 7 | 6 | 11 | 8 |
| P | 7.02 | 11.84 | 15.15 | 8.83 |
| $\beta^{-T+1}$ | $9.54 \times 10^{-7}$ | $1.46 \times 10^{-11}$ | $7.11 \times 10^{-15}$ | $1.49 \times 10^{-8}$ |

| | |
|---|---|
| $\beta$ | = floating-point number base |
| T | = number of base $\beta$ digits in mantissa |
| P | = estimate of the number of significant decimal digits represented by a base $\beta$ floating-point representation with T base $\beta$ digits in its mantissa |
| | $= 1 + (T-1)\log_{10}\beta$ |
| $\beta^{-T+1}$ | = estimate of relative roundoff error with a base $\beta$ floating-point representation and T base $\beta$ digits in the mantissa |

Figure 2: Some Properties of Typical Internal Floating-Point Representations

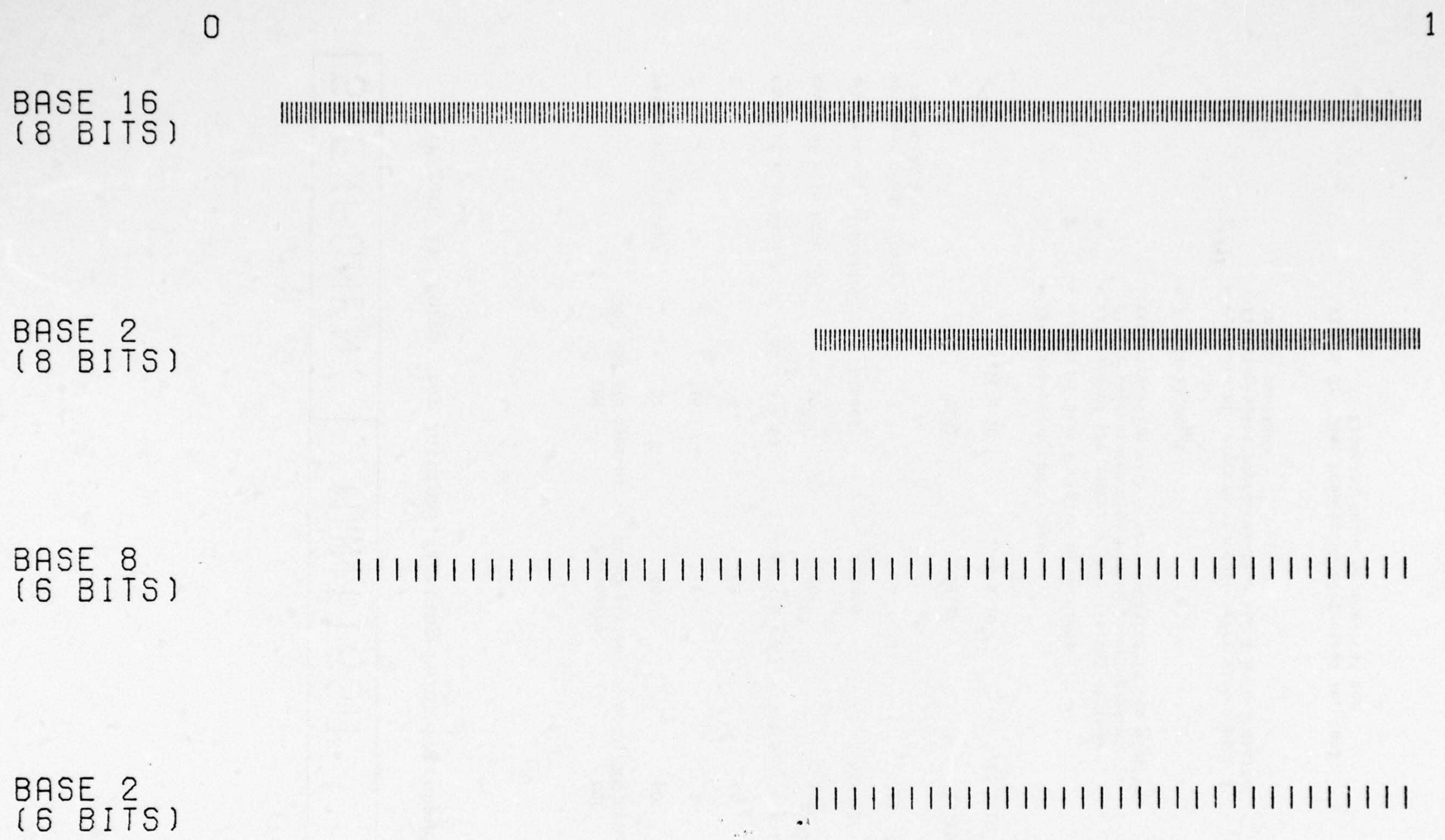


Figure 3: Examples of 8-Bit and 6-Bit Floating-Point Representations in Interval [0,1]

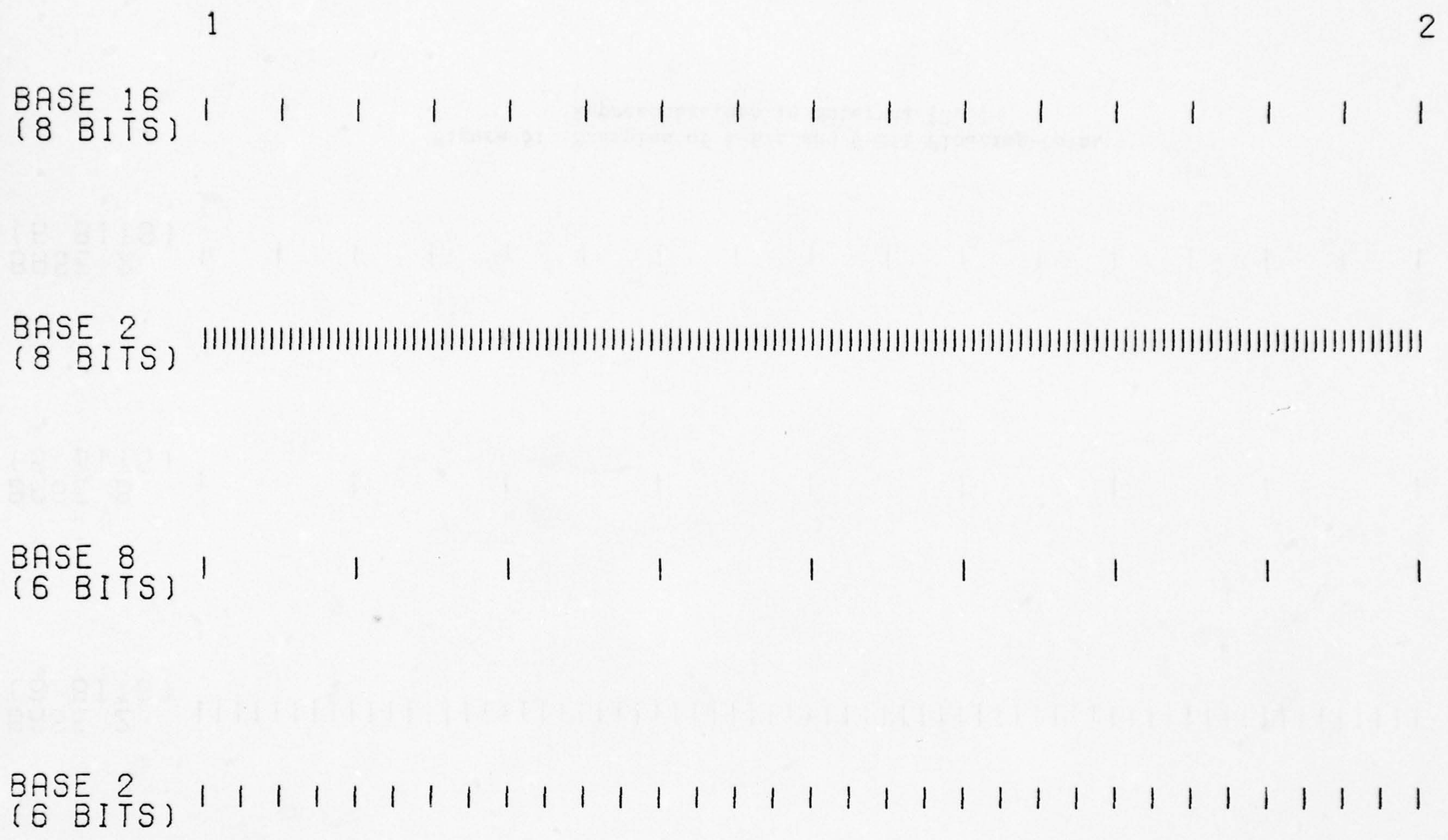


Figure 4: Examples of 8-Bit and 6-Bit Floating-Point Representations in Interval [1,2]

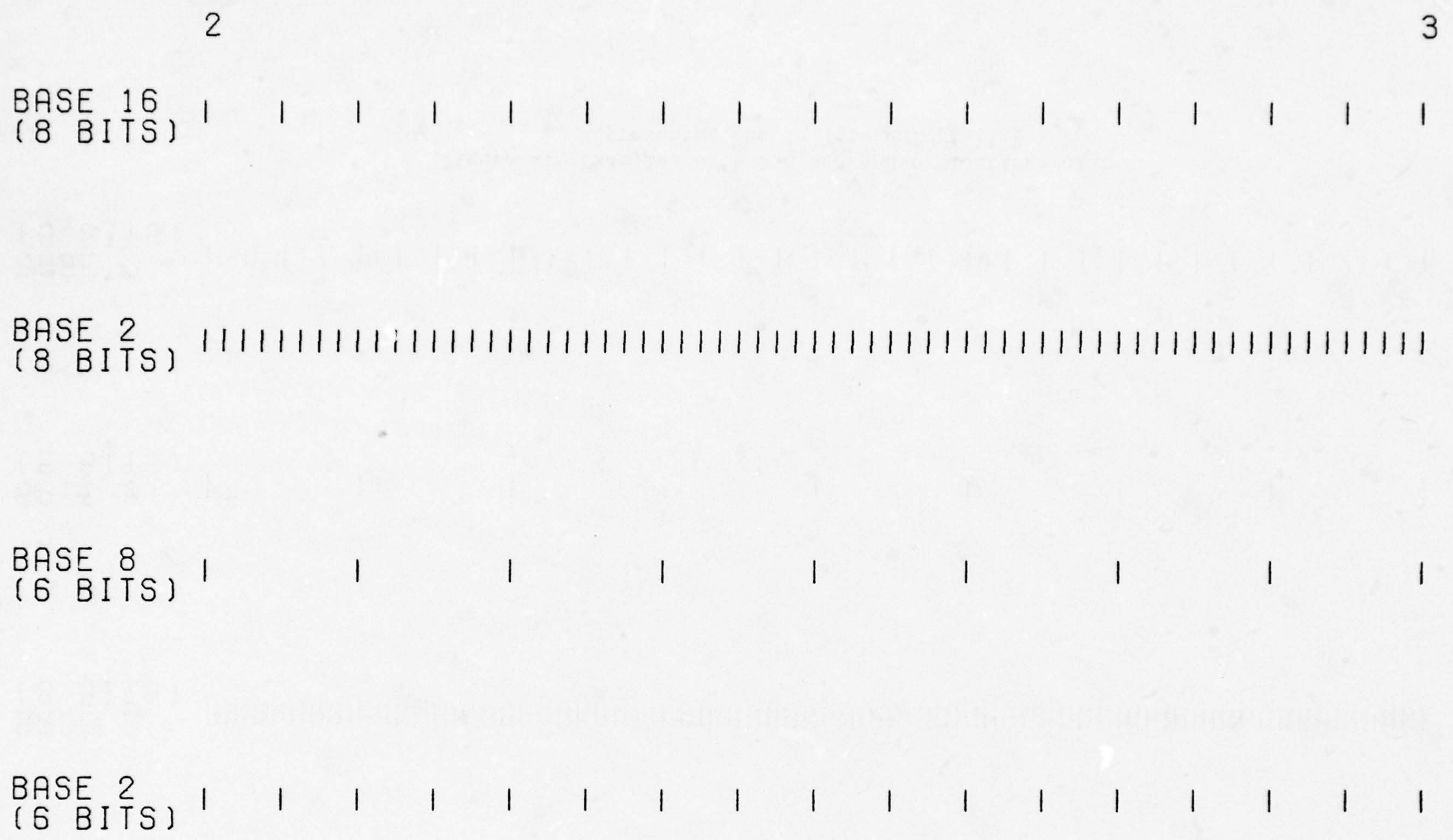


Figure 5: Examples of 8-Bit and 6-Bit Floating-Point Representations in Interval [2,3]

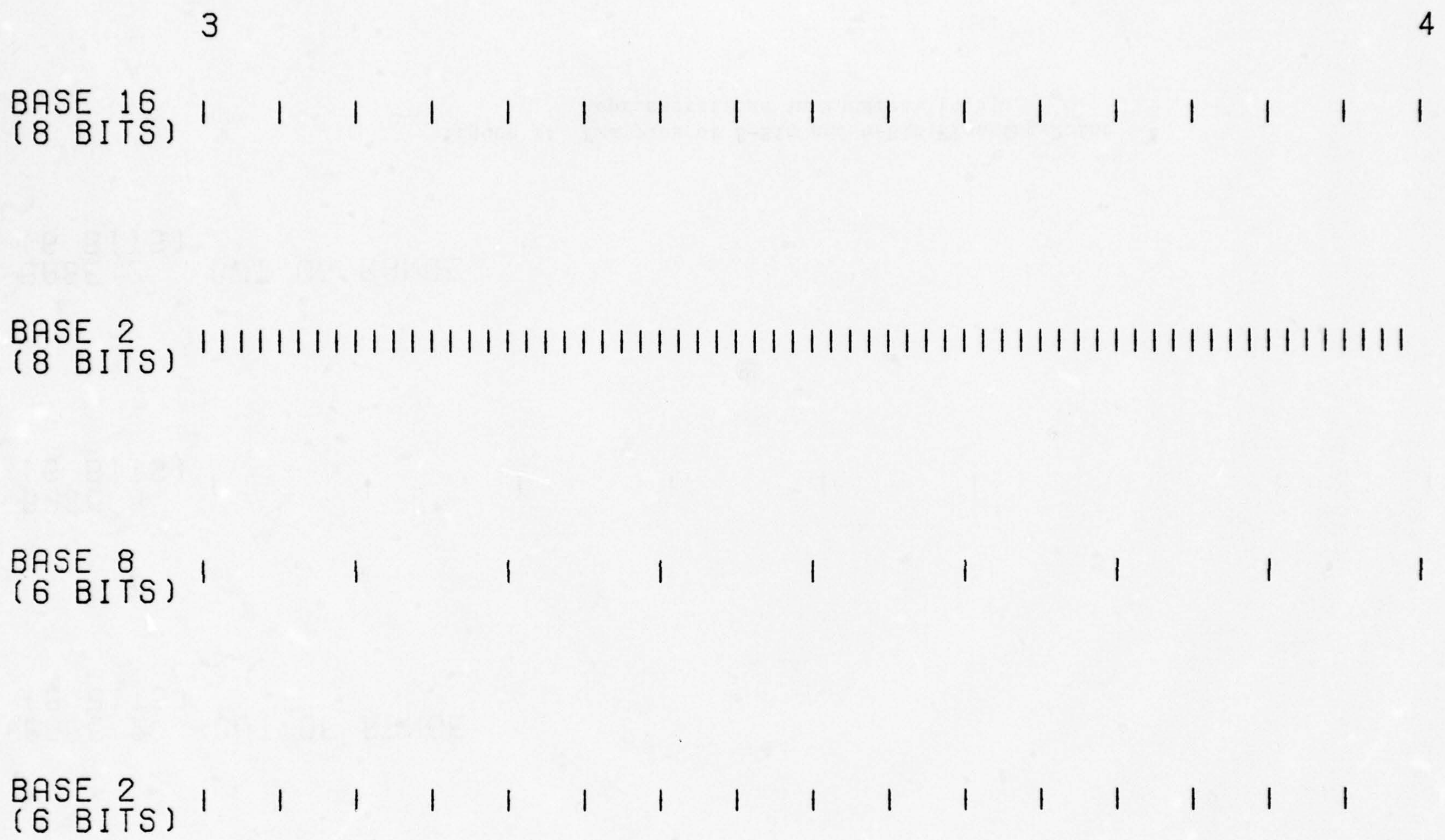


Figure 6: Examples of 8-Bit and 6-Bit Floating-Point Representations in Interval [3,4]

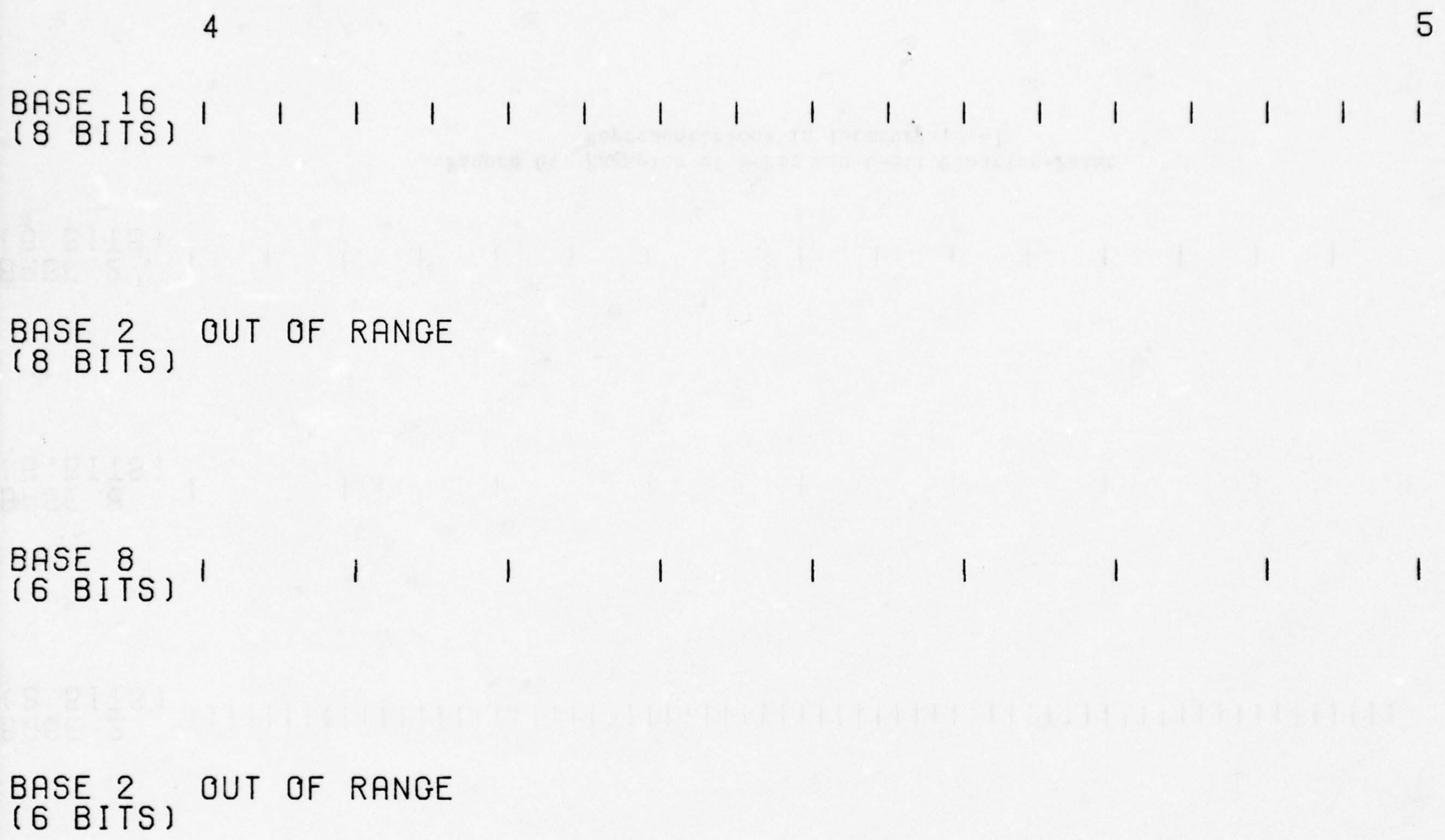


Figure 7: Examples of 8-Bit and 6-Bit Floating-Point Representations in Interval [4,5]

| Interval | Base | Spacing | Number of Points |
|---|---|---|---|
| [0,1] | 16 | $2^{-8}$ * | 242 |
|  | 2 | $2^{-8}$ ** | 130 |
| [1,2] | 16 | $2^{-4}$ | 17 |
|  | 2 | $2^{-7}$ | 129 |
| [2,3] | 16 | $2^{-4}$ | 17 |
|  | 2 | $2^{-6}$ | 65 |
| [3,4] | 16 | $2^{-4}$ | 17 |
|  | 2 | $2^{-6}$ | 64 |
| [4,5] | 16 | $2^{-4}$ | 17 |
|  | 2 | - | 0 |
| [5,16] | 16 | $2^{-4}$ | 177 |
|  | 2 | - | 0 |
| [16,255] | 16 | 1 | 240 |
|  | 2 | - | 0 |

* equal spacing starts at $2^{-4}$

** equal spacing starts at $2^{-1}$

Figure 8: Number of 8-Bit Floating-Point Representations

| Interval | Base | Spacing | Number of Points |
|---|---|---|---|
| [0,1] | 8 | $2^{-6}$ * | 58 |
|  | 2 | $2^{-6}$ ** | 34 |
| [1,2] | 8 | $2^{-3}$ | 9 |
|  | 2 | $2^{-5}$ | 33 |
| [2,3] | 8 | $2^{-3}$ | 9 |
|  | 2 | $2^{-4}$ | 17 |
| [3,4] | 8 | $2^{-3}$ | 9 |
|  | 2 | $2^{-4}$ | 16 |
| [4,5] | 8 | $2^{-3}$ | 9 |
|  | 2 | - | 0 |
| [5,8] | 8 | $2^{-3}$ | 25 |
|  | 2 | - | 0 |
| [8,63] | 8 | 1 | 56 |
|  | 2 | - | 0 |

* equal spacing starts at $2^{-3}$

** equal spacing starts at $2^{-1}$

Figure 9: Number of 6-Bit Floating-Point Representations

There exists one or more values of floating-point numbers A,B, and C such that one or more of the following situations occur

1. Failure of Associative Law

   i.e. $(A+B) + C \neq A + (B+C)$

   $(A*B) * C \neq A * (B*C)$

2. Failure of Distributive Law

   i.e. $A * (B+C) \neq (A*B) + (A*C)$

3. Failure of Cancellation Law

   i.e. $A \neq 0$ and $A*B = A*C$ then $B \neq C$

4. Failure of Closure Law

   i.e. if A and B are exactly representable floating-point numbers then it is possible that the results of A+B, A-B, A*B, or A/B are not exactly representable or undefined

5. Lack of Multiplicative Identity

   i.e. one or more floating-point values of A can exist such that $A*1 \neq A$

6. Floating-Point Values of A and B Can Exist Such that

   $$A * \frac{B}{A} \neq B \text{ for } A \neq 0$$

7. Non-Uniform Distribution of Floating-Point Number Representations

   i.e. if A, B, and C are three successive floating-point numbers then it is possible that $|B-A| \neq |C-B|$ (for examples, see Figures 3,4,5,6,7).

8. Weakening of Inequality Relationships

   $A<B$ does not necessarily imply $A+C<B+C$ but does imply $A+C \leq B+C$

   $A<B$ and $C<D$ does not necessarily imply $A+C<B+D$ but does imply $A+C \leq B+D$

   $B<C$ and $A>0$ does not necessarily imply $A*B<A*C$ but does imply $A*B \leq A*C$

Figure 10: Some Properties of Floating-Point Arithmetic Which Differ from Those for Real Arithmetic (from References I-17, I-27)

```
Program #1                        Program #2

                                  ONE   = 1.0

                                  TWO   = 2.0

                                  THREE = 3.0

                                  FIVE  = 5.0

H = 1.0/2.0                       H = ONE/TWO

X = 2.0/3.0 - H                   X = TWO/THREE - H

Y = 3.0/5.0 - H                   Y = THREE/FIVE - H


                            ⎧ E = (X+X+X) - H
       included in          ⎪
both Programs #1 and #2     ⎨ F = (Y+Y+Y+Y+Y) - H
                            ⎪
                            ⎪ Q = 2.0 * F/E
                            ⎪
                            ⎩ PRINT, Q
```

Exact value of Q = 0.0/0.0

Computed values of Q

| | | CDC 6600 | CDC 7600 |
|---|---|---|---|
| Program #1 | with truncating arithmetic | - 6.0 | - 6.0 |
| | with rounded arithmetic | - 6.0 | - 6.0 |
| Program #2 | with truncating arithmetic | 3.0 | 3.0 |
| | with rounded arithmetic | - 6.0 | 4.0 |

Figure 11: Example of Problem Exhibiting Roundoff Error Propagation and Possible Inconsistency between Results from Compile-Time and Execution-Time Arithmetic (from Reference IV-6)

| Number of Decimal Digits, N | Binary | Octal | Hexadecimal |
|---|---|---|---|
| 1 | 5 | 3(9)* | 2(8)* |
| 2 | 8 | 4(12) | 3(12) |
| 3 | 11 | 5(15) | 4(16) |
| 4 | 15 | 6(18) | 5(20) |
| 5 | 18 | 7(21) | 6(24) |
| 6 | 21 | 8(24) | 6(24) ← IBM 360,370 |
| 7 | 25 ← PDP-10 | 9(27) | 7(28) |
| 8 | 28 | 10(30) | 8(32) |
| 9 | 31 | 11(33) | 9(36) |
| 10 | 35 | 13(39) ← B5500 | 10(40) |
| 11 | 38 | 14(42) | 11(44) |
| 12 | 41 | 15(45) | 11(44) |
| 13 | 45 | 16(48) | 12(48) |
| 14 | 48 ← CDC 6000, 7000 | 17(51) | 13(52) |
| 15 | 51 | 18(54) | 14(56) |
| 16 | 55 | 19(57) | 15(60) |
| 17 | 58 | 20(60) | 16(64) |
| 18 | 61 | 21(63) | 16(64) |
| 19 | 65 | 23(69) | 17(68) |
| 20 | 68 | 24(72) | 18(72) |

*Parenthesized numbers indicate equivalent numbers of bits occupied by corresponding number of base 8 or 16 digits.

Figure 12: Summary of Minimum Number of Digits Needed in Binary, Octal, and Hexadecimal Mantissas So That Rounded In-and-Out Conversions Will Identically Recover Any N-Digit Decimal Number (from Reference I-11)

$$S_I = \sum_{J=1}^{N} A_{IJ} B_J$$

Program #1

```
  S(I) = 0.0

  DO 1 J = 1,N

1 S(I) = S(I) + A(I,J) * B(J)
```

Program #2

```
  DOUBLE PRECISION  SS

  SS = 0.D0

  DO  2  J = 1,N

2 SS = SS + DBLE(A(I,J)) * DBLE(B(J))

  S(I) = SS
```

Test Results on Linear Equation Solvers

AX = B　　　　A is 50 x 50

| Computer/Compiler | Using #1 (Secs) | Using #2 (Secs) |
|---|---|---|
| IBM 370/155 H(Compiler (opt.2) | .76 | 1.88 |
| CDC 6500 RUN Compiler | 1.41 | 8.90 |
| Univac 1106(Exec 8) FORTRAN 5 Compiler | .89 | 1.61 |

Note: In the above cases, Program #1 is never more than one digit less accurate than Program #2

*from T.J. Aird, "Linear Equation Solvers," IMSL Numerical Computations Newsletter, Issue #5,October 1973.

Figure 13: Example of Tradeoff between Accuracy and Time*

Problem: $Y' = -Y,\ Y(0) = 1$

Solution Technique: $Y_{K+1} = Y_K + H * F(X_K + \frac{H}{2}, Y_K + \frac{H}{2} * F(X_K, Y_K))$ (Modified Euler Method)

with step size $H = 2^{-N}$ $N = 1,2,\ldots 10$

| N | Single Precision Error | | Partial Double Precision Error | |
|---|---|---|---|---|
| | IBM 360/67 | DEC PDP-10 | IBM 360/67 | DEC PDP-10 |
| 1 | $2.3 \times 10^{-2}$ | $2.3 \times 10^{-2}$ | $2.3 \times 10^{-2}$ | $2.3 \times 10^{-2}$ |
| 2 | $4.6 \times 10^{-3}$ | $4.6 \times 10^{-3}$ | $4.6 \times 10^{-3}$ | $4.6 \times 10^{-3}$ |
| 3 | $1.1 \times 10^{-3}$ | $1.1 \times 10^{-3}$ | $1.1 \times 10^{-3}$ | $1.1 \times 10^{-3}$ |
| 4 | $2.5 \times 10^{-4}$ | $2.5 \times 10^{-4}$ | $2.5 \times 10^{-4}$ | $2.5 \times 10^{-4}$ |
| 5 | $6.1 \times 10^{-5}$ | $6.1 \times 10^{-5}$ | $6.1 \times 10^{-5}$ | $6.1 \times 10^{-5}$ |
| 6 | $1.4 \times 10^{-5}$ | $1.5 \times 10^{-5}$ | $1.5 \times 10^{-5}$ | $1.5 \times 10^{-5}$ |
| 7 | $1.6 \times 10^{-6}$ | $3.8 \times 10^{-6}$ | $3.8 \times 10^{-6}$ | $3.8 \times 10^{-6}$ |
| 8 | $-3.1 \times 10^{-6}$ | $9.6 \times 10^{-7}$ | $9.7 \times 10^{-7}$ | $9.4 \times 10^{-7}$ |
| 9 | $-8.2 \times 10^{-6}$ | $2.6 \times 10^{-7}$ | $2.7 \times 10^{-7}$ | $2.3 \times 10^{-7}$ |
| 10 | $-1.6 \times 10^{-5}$ | $5.4 \times 10^{-8}$ | $9.4 \times 10^{-8}$ | $5.8 \times 10^{-8}$ |

Figure 14: Example of Effect of Roundoff Error in Single and Partial Double Precision (from Reference II-30)

| Argument Range | | Accuracy for Random Arguments Frequency of N-Bit Error | | | | Maximum Relative Error | RMS Error |
|---|---|---|---|---|---|---|---|
| X | Y | 0 | 1 | 2 | 3 | | |
| (1/16, 16) | (-4, 4) | 1933 | 67 | 0 | 0 | $.456 \times 10^{-6}$ | $.119 \times 10^{-6}$ |
| $(2^{-16}, 2^{16})$ | (-16, 16) | 1771 | 226 | 3 | 0 | $.483 \times 10^{-6}$ | $.124 \times 10^{-6}$ |
| $(2^{-32}, 2^{32})$ | (-8, 8) | 1906 | 94 | 0 | 0 | $.459 \times 10^{-6}$ | $.116 \times 10^{-6}$ |
| $(2^{-64}, 2^{64})$ | (-4, 4) | 1938 | 62 | 0 | 0 | $.442 \times 10^{-6}$ | $.111 \times 10^{-6}$ |
| $(2^{-8}, 2^{8})$ | (-32, 32) | 1609 | 347 | 44 | 0 | $.561 \times 10^{-6}$ | $.136 \times 10^{-6}$ |
| $(2^{-4}, 2^{4})$ | (-64, 64) | 1392 | 440 | 140 | 28 | $.633 \times 10^{-6}$ | $.175 \times 10^{-6}$ |

Figure 15: Example of Argument Testing Results for a Subroutine to Compute X**Y (from Reference IV-5)

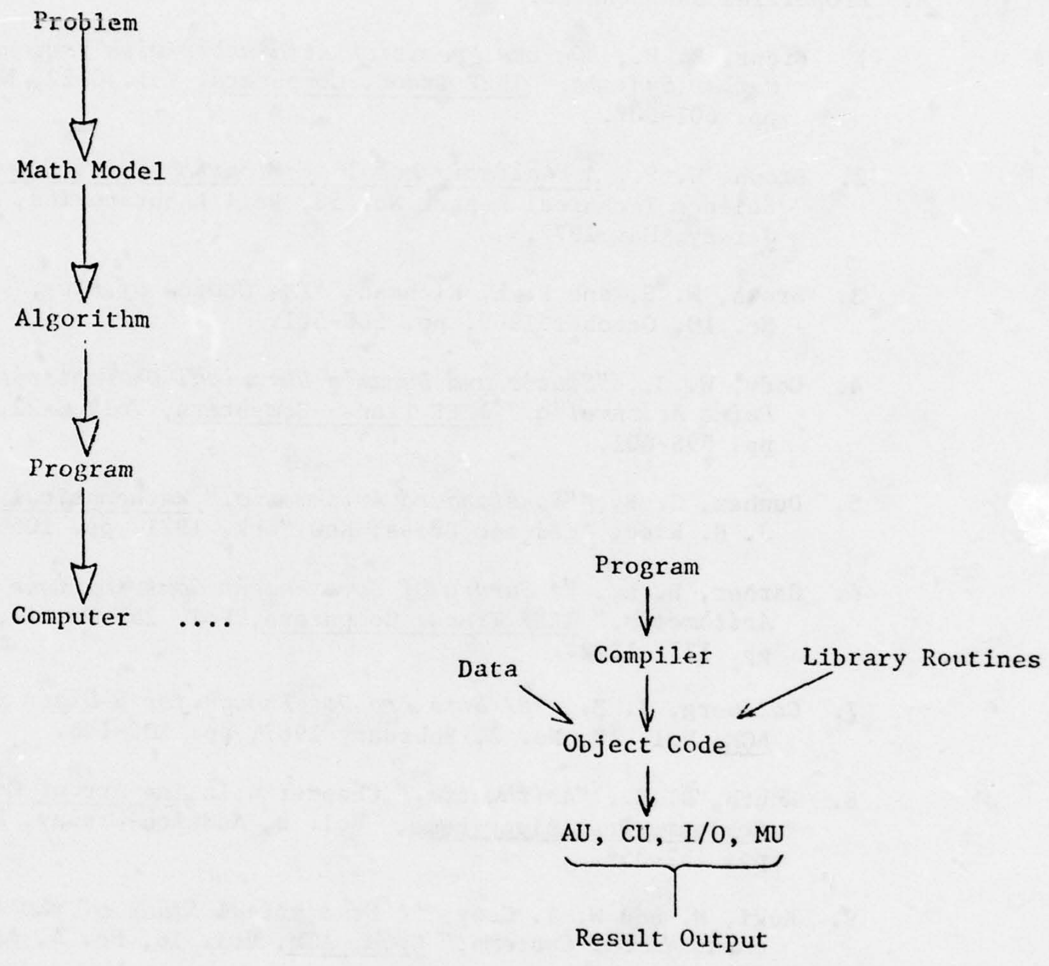


Figure 16: Sources of Errors

## APPENDIX

*A Guide to the Literature on Floating-Point Arithmetic and Its Influences on Mathematical Software*

## I. FLOATING-POINT ARITHMETIC

### A. Properties and Behavior

1. Brent, R. P., *"On the Precision Attainable with Various Floating Point Number Systems,"* IEEE Trans. Computers, Vol. C-22, No. 6, June 1973, pp. 601-607.

2. Brown, W. S., *A Realistic Model of Floating-Point Computation*, Computing Science Technical Report No. 58, Bell Laboratories, Murray Hill, New Jersey, May 1977.

3. Brown, W. S. and P. L. Richman, *"The Choice of Base,"* Comm. ACM, Vol. 12, No. 10, October 1969, pp. 560-561.

4. Cody, W. J., *"Static and Dynamic Numerical Characteristics of Floating-Point Arithmetic,"* IEEE Trans. Computers, Vol. C-22, No. 6, June 1973, pp. 598-601.

5. Dunham, C. B., *"Nonstandard Arithmetic,"* Mathematical Software, edited by J. R. Rice, Academic Press, New York, 1971, pp. 105-111.

6. Garner, H. L., *"A Survey of Some Recent Contributions to Computer Arithmetic,"* IEEE Trans. Computers, Vol. 25, No. 12, December 1976, pp. 1277-1282.

7. Goldberg, I. B., *"27 Bits Are Not Enough for 8-Digit Accuracy,"* Comm. ACM, Vol. 10, No. 2, February 1967, pp. 105-106.

8. Knuth, D. E., *"Arithmetic,"* Chapter 4 in The Art of Computer Programming: Seminumerical Algorithms, Vol. 2, Addison-Wesley, Reading, Mass., 1969, pp. 162-359.

9. Kuki, H. and W. J. Cody, *"A Statistical Study of the Accuracy of Floating Point Number Systems,"* Comm. ACM, Vol. 16, No. 4, April 1973, pp. 223-230.

10. Matula, D. W., *"A Formalization of Floating-Point Numeric Base Conversion,"* IEEE Trans. Computers, Vol. C-19, No. 8, August 1970, pp. 681-692.

11. Matula, D. W., *"In-and-Out Conversions,"* Comm. ACM, Vol. 11, No. 1, January 1968, pp. 47-50.

12. Matula, D. W., *"Significant Digits: Numerical Analysis or Numerology,"* Proc. IFIPS - 71, North-Holland, 1972, pp. 1278-1283.

13. *Proceedings of the Third Symposium on Computer Arithmetic*, IEEE, New York, 1975.

14. Richman, P., *Floating-Point Number Representations: Base Choice Versus Exponent Range*, Technical Report No. CS64, Computer Science Department, Stanford University, Stanford, California, April 1967.

15. *"Special Issue on Computer Arithmetic,"* IEEE Trans. Computers, Vol. C-19, No. 8, August 1970, pp. 679-757.

16. *"Special Section on Computer Arithmetic,"* IEEE Trans. Computers, Vol. C-22, No. 6, June 1973, pp. 549-607.

17. Sterbenz, P. H., *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

18. Sweeney, D. W., *"An Analysis of Floating Point Addition,"* IBM Systems J., Vol. 4, No. 1, 1965, pp. 31-42.

19. Thacher, H. C., *"Making Special Arithmetic Available,"* Mathematical Software, edited by J. R. Rice, Academic Press, New York, 1971, pp. 113-119.

20. Tienari, M., *"On the Control of Floating Point Mantissa Length in Iterative Computations,"* Proc., International Computing Symposium 1973, North-Holland, Amsterdam, 1974, pp. 315-322.

21. Yohe, J. M., *Accurate Conversion between Number Bases*, Technical Summary Report No. 1109, Mathematics Research Center, University of Wisconsin, Madison, October 1970.

22. Yohe, J. M., *Best Possible Floating Point Arithmetic*, Technical Summary Report No. 1054, Mathematics Research Center, University of Wisconsin, Madison, Wisconsin, March 1970.

23. Yohe, J. M., *Foundations of Floating Point Computer Arithmetic*, Technical Summary Report No. 1302, Mathematics Research Center, University of Wisconsin, Madison, Wisconsin, January 1973.

24. Yohe, J. M., *"Roundings in Floating-Point Arithmetic,"* IEEE Trans. Computers, Vol. C-22, No. 6, June 1973, pp. 577-586.

25. Young, D. M. and A. E. McDonald, *"On the Surveillance and Control of Number Range and Accuracy in Numerical Computation,"* Proc., Information Processing 68, Vol. 1, North-Holland, Amsterdam, 1969, pp. 145-152.

B. Hardware and Software Considerations

26. Cody, W. J., *"Desirable Hardware Characteristics for Scientific Computation,"* ACM SIGNUM Newsletter, Vol. 2, No. 1, January 1971, pp. 16-31.

27. Cody, W. J., *"The Influence of Machine Design on Numerical Algorithms,"* Proc., Spring Joint Computer Conference, Vol. 30, AFIPS Press, Montvale, New Jersey, 1967, pp. 305-309.

28. Gear, C. W., *What Do We Need in Programming Languages for Mathematical Software?*, Report No. UIUCDCS-R-74-652, Department of Computer Science, University of Illinois, Urbana, Illinois, 1974.

29. Parlett, B. N. and Y. Wang, *"The Influence of the Compiler on the Cost of Mathematical Software - in Particular on the Cost of Triangular Factorization,"* ACM Trans. Math. Software, Vol. 1, No. 1, March 1975, pp. 35-46.

30. Redish, K. A. and W. Ward, *"Environment Enquiries for Numerical Analysis,"* ACM SIGNUM Newsletter, Vol. 6, No. 1, January 1971, pp. 10-15.

31. Ris, F. N., *"A Unified Decimal Floating-Point Architecture for the Support of High-Level Languages (extended abstract),"* ACM SIGNUM Newsletter, Vol. 11, No. 3, October 1976, pp. 18-23.

32. Shriver, B. D.,*"Microprogramming and Numerical Analysis,"* IEEE Trans. Computers, Vol. C-20, No. 7, July 1971, pp. 808-811.

33. Turner, L. R., *Hardware (or Software) Provisions for Multiple Precision Floating-Point Arithmetic,* Technical Memo TM X-1885, NASA Lewis Research Center, Cleveland, Ohio, September 1969.

C. Error Analysis

34. Chai, A. S., *Statistical Estimation of the Effect of Initial and Roundoff Errors in Digital Computation,* Ph.D. Thesis, Electrical Engineering Department, University of Wisconsin, Madison, 1967.

35. Kahan, W., *"A Survey of Error Analysis,"* Proc., IFIP Congress 71, North-Holland, Amsterdam, 1972, pp. 1214-1239.

36. Kaneko, T. and B. Liu, *"On Local Roundoff Errors in Floating-Point Arithmetic,"* J. Assoc. Comput. Mach., Vol. 20, No. 3, July 1973, pp. 391-398.

37. Marasa, J. D. and D. W. Matula, *"A Simulative Study of Correlated Error Propagation in Various Finite-Precision Arithmetics,"* IEEE Trans. Computers, Vol. C-22, No. 6, June 1973, pp. 587-597.

38. McKeeman, W. M., *"Representation Error for Real Numbers in Binary Computer Arithmetic,"* IEEE Trans. Electronic Computers, Vol. EC-16, No. 5, October 1967, pp. 682-683.

39. Miller, W., *"Software for Roundoff Analysis,"* ACM Trans. Math. Software, Vol. 1, No. 2, June 1975, pp. 108-128.

40. Miller, W. and D. Spooner, *Software for Roundoff Analysis, II,* Report, Computer Science Department, The Pennsylvania State University, University Park, Pennsylvania, August 1975.

41. Rall, L. B. (ed), *Error in Digital Computation,* Vols. 1 and 2, J. Wiley, New York, 1965.

42. Richman, P. L., *"Automatic Error Analysis for Determining Precision,"* Comm. ACM, Vol. 15, No. 9, September 1972, pp. 813-817.

43. Schoenfeld, L., *Floating Point Error Estimates,* Technical Summary Report No. 721, Mathematics Research Center, University of Wisconsin, Madison, Wisconsin, August 1967.

44. Stoutemyer, D. R., *"Automatic Error Analysis Using Computer Algebraic Manipulation,"* ACM Trans. Math. Software, Vol. 3, No. 1, March 1977, pp. 26-43.

45. Tienari, M., *"A Statistical Model of Roundoff Error for Varying Length Floating-Point Arithmetic,"* BIT, Vol. 10, No. 3, March 1970, pp. 355-365.

46. Tsao, N. K., *"On the Distribution of Significant Digits and Roundoff Errors,"* Comm. ACM, Vol. 17, No. 5, May 1974, pp. 269-271.

47. Wilkinson, J. H., *Rounding Errors in Algebraic Processes,* Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

48. Wilkinson, J. H., *"Modern Error Analysis,"* SIAM Rev., Vol. 13, No. 4, October 1971, pp. 548-568.

D. Pocket Calculators, Microcomputers, and Floating-Point

49. Cody, W. J., *"The Challenge in Numerical Software for Microcomputers,"* this proceedings.

50. Kahan, W., *And Now for Something Completely Different: The Texas Instruments SR-52,* Memorandum No. UCB/ERL M77/23, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, California, April 1977.

51. Kahan, W. and B. N. Parlett, *Can you Count on Your Calculator,* Memorandum No. UCB/ERL M77/21, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, California, April 1977.

52. Linker, S., *"What's in a Floating Point Package?,"* BYTE, Vol. 2, No. 5, May 1977, pp. 62-66.

53. Maples, M. D., *Floating Point Package for INTEL 8008 and 8080 Micro-Processors,* Report No. UCRL-51940, Lawrence Livermore Laboratory, Livermore, California, 1975.

54. Osofsky, B. L., *"Small Calculators for the Mathematician,"* Notices of the Amer. Math. Society, Vol. 23, No. 8, December 1976, pp. 442-430.

55. Stakem, P. H., *"Using a Calculator Chip to Extend a Microprocessor's Capabilities,"* Computer Design, Vol. 14, No. 9, September 1975, pp. 98-99.

II. ALGORITHMIC IMPLEMENTATION CONSIDERATIONS

1. Aird, T. J., *"More on Single and Double Accumulation,"* IMSL Numerical Computations Newsletter, Issue 5, October 1973, p. 3.

2. Clark, N. W. and W. J. Cody, *"Self-Contained Exponentiation,"* Proc., Fall Joint Computer Conference, Vol. 35, AFIPS Press, Montvale, New Jersey, 1969, pp. 701-706.

3. Cody, W. J., *"The FUNPACK Package of Special Function Subroutines,"* ACM Trans. Math. Software, Vol. 1, No. 1, March 1975, pp. 13-25.

4. Cody, W. J., *"An Overview of Software Development for Special Functions,"* Numerical Analysis - Dundee 1975, Lecture Notes in Mathematics, Vol. 506, edited by G. A. Watson, Springer-Verlag, Berlin, 1976, pp. 38-48.

5. Cody, W. J., *"Software for the Elementary Functions,"* Mathematical Software, edited by J. R. Rice, Academic Press, New York, 1971, pp. 171-185.

6. Evans, D. J. (ed.), *Software for Numerical Mathematics*, Academic Press, London, 1974.

7. Eve, J., *"The Evaluation of Polynomials,"* Numer. Math, Vol. 6, No. 1, October 1964, pp. 17-21.

8. Fike, C. T., *Computer Evaluation of Mathematical Functions*, Prentice-Hall, Englewood Cliffs, New Jersey, 1968.

9. Forsythe, G. E., *"Pitfalls in Computation, or Why a Math Book Isn't Enough,"* Amer. Math. Monthly, Vol. 77, No. 9, November 1970, pp. 931-956.

10. Forsythe, G. E., *"What Is a Satisfactory Quadratic Equation Solver?,"* Constructive Aspects of the Fundamental Theorem of Algebra, edited by B. Dejon and P. Henrici, Wiley-Interscience, New York, 1969, pp. 53-61.

11. Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

12. Gregory, J., *"A Comparison of Floating Point Summation Methods,"* Comm. ACM, Vol. 15, No. 9, September 1972, p. 838.

13. Hart, J. et al., *Computer Approximations*, J. Wiley, New York, 1968.

14. Kahan, W., *"Further Remarks on Reducing Truncation Errors,"* Comm. ACM, Vol. 8, No. 1, January 1965, p. 40.

15. Kahan, W., *Implementation of Algorithms, Parts I and II*, Technical Report No. 20, Department of Computer Science, University of California, Berkeley, 1973; available as AD-769124 from National Technical Informatio Service, Springfield, Virginia.

16. Knuth, D. E., *"Evaluation of Polynomials by Computers,"* Comm. ACM, Vol. 5, No. 12, December 1962, pp. 595-599.

17. Kuki, H., *"Mathematical Function Subprograms for Basic System Libraries - Objectives, Constraints, and Trade-Off,"* Mathematical Software, edited by J. R. Rice, Academic Press, New York, 1971, pp. 187-199.

18. Kuki, H. and J. Ascoly, *"FORTRAN Extended-Precision Library,"* IBM Systems J., Vol. 10, No. 1, 1971, pp. 39-61.

19. Lawson, C. L. and R. J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

20. Linnainmaa, S., *"Analysis of Some Known Methods of Improving the Accuracy of Floating-Point Sums,"* BIT, Vol. 14, No. 2, 1974, pp. 167-202.

21. Linz, P., *"Accurate Floating-Point Summation,"* Comm. ACM, Vol. 13, No. 6, June 1970, pp. 361-362.

22. Luke, Y. L., *Mathematical Functions and Their Approximation*, Academic Press, New York, 1975.

23. Lyness, J. N., *"The Effect of Inadequate Convergence Criteria in Automatic Routines,"* Comput. J., Vol. 12, No. 3, August 1969, pp. 279-281.

24. Lyness, J. M., *"Guidelines for Automatic Quadrature Routines,"* Proc., IFIP Congress 71, Vol. 2, North-Holland, Amsterdam, 1972, pp. 1351-1355.

25. Lyness, J. N. and J. J. Kaganove, *"Comments on the Nature of Automatic Quadrature Routines,"* ACM Trans. Math Software, Vol. 2, No. 1, March 1976, pp. 65-81.

26. Malcolm, M. A., *"On Accurate Floating-Point Summation,"* Comm. ACM, Vol. 14, No. 11, November 1971, pp. 731-736.

27. Paul, G. and M. W. Wilson, *"Should the Elementary Function Library Be Incorporated into Computer Instruction Sets,"* ACM Trans. Math. Software, Vol. 2, No. 2, June 1976, pp. 132-142.

28. Peters, G. and J. H. Wilkinson, *"Practical Problems Arising in the Solution of Polynomial Equations,"* J. Inst. Maths. Applics., Vol. 8, No. 1, August 1971, pp. 16-35.

29. Schmid, H., *Decimal Computation*, J. Wiley, New York, 1974.

30. Shampine, L. F. and R. C. Allen, *Numerical Computing: An Introduction*, W. B. Saunders, Philadelphia, Pennsylvania, 1973.

31. Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: The Initial Value Problem*, Freeman, San Francisco, California, 1975.

32. Stegun, I. A. and M. Abramowitz, *"Pitfalls in Computation,"* SIAM J., Vol. 4, No. 4, December 1956, pp. 207-219.

33. Walters, J. S., *"A Unified Algorithm for Elementary Functions,"* Proc., Spring Joint Computer Conference, Vol. 38, AFIPS Press, Montvale, New Jersey, 1971, pp. 379-385.

34. Wolfe, J. M., *"Reducing Truncation Errors by Programming,"* Comm. ACM, Vol. 7, No. 6, June 1964, pp. 355-356.

35. Yarborough, L., *"Precision Calculation of e and π Constants,"* Comm. ACM, Vol. 10, No. 9, September 1967, p. 537.

III. ERROR-MONITORING TECHNIQUES

A. Interval Analysis

1. Chartres, B. A., *"Automatic Controlled Precision Calculations,"* J. Assoc. Comput. Mach., Vol. 13, No. 3, July 1966, pp. 386-403.

2. Ginsberg, M., *Introduction to the Study of Algorithms for Computing Upper and Lower Bounds to the Exact Solution of Problems in Numerical Analysis,* Technical Report No. CP 73024, Department of Computer Science, Southern Methodist University, Dallas, Texas, September 1973.

3. Hansen, E. (ed.), *Topics in Interval Analysis,* Oxford University Press, London, 1969.

4. Moore, R. E., *Interval Analysis,* Prentice-Hall, Englewood Cliffs, New Jersey, 1966.

5. Nickel, K. (ed.), *Interval Mathematics,* Vol. 29, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1975.

6. Richman, P. L., *Variable - Precision Interval Arithmetic,* Technical Memorandum No. MM-69-1374-26, Bell Telephone Laboratories, Murray Hill, New Jersey, November 1969.

B. Multiple Precision

7. Brent, R. P., *A FORTRAN Multiple-Precision Arithmetic Package,* Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1976.

8. Dekker, T. J., *"A Floating-Point Technique for Extending the Available Precision,"* Numer. Math., Vol. 18, No. 3, 1971, pp. 224-242.

9. Hull, T. E. and J. J. Hofbauer, *Language Facilities for Multiple Precision Floating Point Computation, with Examples and the Description of a Preprocessor,* Technical Report No. 63, Department of Computer Science, University of Toronto, Ontario, Canada, February 1974.

10. Knuth, D. E., *"Multiple - Precision Arithmetic,"* Section 4.3, The Art of Computer Programming: Seminumerical Algorithms, Vol. 2, Addison-Wesley, Reading, Mass., 1969, pp. 229-280.

11. Tienari, M., *Varying Length Floating Point Arithmetic: A Necessary Tool for the Numerical Analyst,* Technical Report No. 62, Computer Science Department, Stanford University, Stanford, California, 1967.

12. Wyatt, W. T., Jr., D. W. Lozier, and D. J. Orser, *"A Portable Extended Precision Arithmetic Package and Library with Fortran Precompiler,"* ACM Trans. Math Software, Vol. 2, No. 3, September 1976, pp. 209-231.

C. Significance Arithmetic

13. Ashenhurst, R. L., *"Experimental Investigation of Unnormalized Arithmetic,"* Error in Digital Computation, Vol. 2, edited by L. B. Rall, J. Wiley, New York, 1965, pp. 3-37.

14. Ashenhurst, R. L., *"Number Representation and Significance Monitoring,"* Mathematical Software, edited by J. R. Rice, Academic Press, New York, 1971, pp. 67-92.

15. Ashenhurst, R. L. and N. Metropolis, *"Error Estimation in Computer Calculation,"* Amer. Math. Monthly, Vol. 72, No. 2, (part II), February 1965, pp. 47-58.

IV. PERFORMANCE TESTING OF MATHEMATICAL SOFTWARE

1. Bailey, C. B. and R. E. Jones, *"Usage and Argument Monitoring of Mathematical Library Routines,"* ACM Trans. Math. Software, Vol. 1, No. 3, September 1975, pp. 196-209.

2. Barinka, L. L., *"Some Experience with Constructing, Testing, and Certifying a Standard Mathematical Subroutine Library,"* ACM Trans. Math. Software, Vol. 1, No. 2, June 1975, pp. 165-177.

3. Bright, H. S. and I. J. Cole, *"A Method of Testing Programs for Data Sensitivity,"* Program Test Methods, edited by W. C. Hetzel, Prentice-Hall, Englewood Cliffs, New Jersey, 1973, pp. 143-162.

4. Cody, W. J., *"The Evaluation of Mathematical Software,"* Program Test Methods, edited by W. C. Hetzel, Prentice-Hall, Englewood Cliffs, New Jersey, 1973, pp. 121-133.

5. Cody, W. J., *"Performance Testing of Function Subroutines,"* Proc., Spring Joint Computer Conference, Vol. 34, AFIPS Press, Montvale, New Jersey, 1969, pp. 759-763.

6. Dorr, F. W. and C. B. Moler, *"Roundoff Error on the CDC 6600/7600 Computers,"* ACM SIGNUM Newsletter, Vol. 8, No. 2, April 1973, pp. 24-26.

7. Enright, W. H., R. Bedet, I. Farkas, and T. E. Hull, *"Comparing Numerical Methods for Stiff Systems of O.D.E.'s,"* BIT, Vol. 15, No. 1, 1975, pp. 10-48.

8. Ginsberg, M., *"Testing Techniques for Evaluating the Numerical Behavior of Mathematical Software,"* Proceedings, Third Annual Computer Science Conference of the Federation of North Texas Area Universities, North Texas State University, Denton, Texas, 1976, pp. 173-190.

9. Hammer, C., *"Statistical Validation of Mathematical Computer Routines,"* Proc., Spring Joint Computer Conference, Vol. 30, AFIPS Press, Montvale, New Jersey, 1967, pp. 331-333.

10. Hetzel, W. C. (ed.), *Program Test Methods*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973, pp. 313-348.

11. Hull, T. E., *"The Validation and Comparison of Programs for Stiff Systems,"* Stiff Differential Equations, edited by R. A. Willoughby, Plenum Press, New York, 1974, pp. 151-164.

12. Hull, T. E., W. H. Enright, B. M. Fellen, and A. E. Sedgwick, *"Comparing Numerical Methods for Ordinary Differential Equations,"* SIAM J. Numer. Anal., Vol. 9, No. 4, December 1972, pp. 603-637.

13. Jenkins, M. A. and J. F. Traub, *"Principles for Testing Polynomial Zerofinding Programs,"* ACM Trans. Math. Software, Vol. 1, No. 1, March 1975, pp. 26-34.

14. Kahan, W., *"A Problem,"* ACM SIGNUM Newsletter, Vol. 6, No. 3, November 1971, p. 6.

15. Krogh, F. T., *"On Testing a Subroutine for the Numerical Integration of Ordinary Differential Euqations,"* J. Assoc. Comput. Mach., Vol. 20, No. 4, October 1973, pp. 545-562.

16. Lozier, D. W., L. C. Maximon, and W. L. Sadowski, *"A Bit Comparison Program for Algorithm Testing,"* Comput. J., Vol. 16, No. 2, May 1973, pp. 111-117.

17. Lozier, D. W., L. C. Maximon, and W. L. Sadowski, *"Performance Testing of a FORTRAN Library of Mathematical Function Routines - A Case Study in the Application of Testing Techniques,"* J. Res. Nat. Bur. Standards, Vol. 77B, Nos. 3 and 4, July - December 1973, pp. 101-110.

18. Miller, W., *"Computer Search for Numerical Instability,"* J. Assoc. Comput. Mach., Vol. 22, No. 4, October 1975, pp. 512-521.

19. Newbery, A. C. R. and A. P. Leigh, *"Consistency Tests for Elementary Functions,"* Proc., Fall Joint Computer Conference, Vol. 39, AFIPS Press, Montvale, New Jersey 1971, pp. 419-422.

20. Shampine, L. F., H. A. Watts, and S. M. Davenport, *Solving Non-Stiff Ordinary Differential Equations - The State of the Art*, Report No. SAND-75-0182, Sandia Laboratories, Albuquerque, New Mexico, March 1975.

## V. PORTABILITY CONSIDERATIONS FOR MATHEMATICAL SOFTWARE

1. Aird, T. J. et al., *"Name Standardization and Value Specification for Machine Dependent Constants,"* ACM SIGNUM Newsletter, Vol. 9, No. 4, October 1974, pp. 11-13.

2. Aird, T. J., E. L. Battiste, and W. C. Gregory, *"Portability of Mathematical Software Coded in FORTRAN,"* ACM Trans. Math. Software, Vol. 3, No. 2, June 1977, pp. 113-127.

3. Blue, J. L., *A Portable FORTRAN Program to Find the Euclidean Norm of a Vector*, Computing Science Technical Report No. 45, Bell Laboratories, Murray Hill, New Jersey, February 1977.

4. Cody, W. J., *"The Construction of Numerical Subroutine Libraries,"* SIAM Rev., Vol. 16, No. 1, January 1974, pp. 36-46.

5. Cody, W. J., *"Machine Parameters for Numerical Analysis,"* Proceedings of the NSF/ERDA Workshop on Portability of Numerical Software, edited by W. R. Cowell, Lecture Notes in Computer Science, Springer-Verlag, New York, 1977.

6. Cowell, W. R. (ed.), *Proceedings of the NSF/ERDA Workshop on Portability of Numerical Software*, Lecture Notes in Computer Science, Springer-Verlag, New York, 1977.

7. Ford, B. and Sayers, D. K., *"Developing a Single Numerical Algorithms Library for Different Machine Ranges,"* ACM Trans. Math. Software, Vol. 2, No. 2, June 1976, pp. 115-131.

8. Fox, P. A., A. D. Hall, and N. L. Schryer, *The PORT Mathematical Subroutine Library*, Computing Science Technical Report No. 47, Bell Laboratories, Murray Hill, New Jersey, May 1977.

9. Gentleman, W. M. and S. B. Marovich, *"More on Algorithms that Reveal Properties of Floating Point Arithmetic Units,"* Comm. ACM, Vol. 17, No. 5, May 1974, pp. 276-277.

10. George, J. E., *"Algorithms to Reveal the Representation of Characters, Integers, and Floating-Point Numbers,"* ACM Trans. Math. Software, Vol. 1, No. 3, September 1975, pp. 210-216.

11. Hague, S. J. and B. Ford, *"Portability - Prediction and Correction,"* Software-Practice and Experience, Vol. 6, No. 1, 1976, pp. 61-69.

12. Malcolm, M. A., *"Algorithms to Reveal Properties of Floating-Point Arithmetic,"* Comm. ACM, Vol. 15, No. 11, November 1972, pp. 949-951.

13. Schonfelder, J. L., *"The Production of Special Function Routines for a Multi-Machine Library,"* Software-Practice and Experience, Vol. 6,. No. 1, 1976, pp. 71-82.

14. Smith, B. T., J. M. Boyle, and W. J. Cody, *"The NATS Approach to Quality Software,"* Software for Numerical Mathematics, edited by D. J. Evans, Academic Press, London 1974, pp. 393-405.

## VI. BIBLIOGRAPHIES

1. Bierbaum, F., Intervall - Mathematik. Eine Literaturübersicht, Report No. 74/2, Institut für Praktische Mathematik, University of Karlsruhe, Karlsruhe, Germany, 1974.

2. Edberg, E. and J. Johansson, *"An Annotated Bibliography on Mathematical Software and Related Topics,"* ACM SIGNUM Newsletter, Vol. 11, No. 2, August 1976, pp. 9 - 16 and No. 3, October 1976, p. 6.

3. Einarsson, B., *Bibliography on Numerical Software,* Memorandum No. UCB/ERL M77/19, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, California, March 1977.

4. Ginsberg, M., *"A Guide to the Literature of Modern Numerical Mathematics,"* Bibliography 36, Computing Reveiws, Vol. 16, No. 2, February 1975, pp. 83-97

5. Grooms, D. W., *Computer Software Transferability and Portability,* No. NTIS/PS-76/0388/9WC, National Technical Information Service, Springfield, Virginia, May 1976.

6. Hetzel, W. C. (ed.), *"Bibliography,"* Chapter 9, Program Test Methods, Prentice-Hall, Englewood Cliffs, New Jersey, 1973, pp. 313-348.

7. Sterbenz, P. H., *"Bibliography,"* Floating-Point Computation, Prentice-Hall, Englewood Cliffs, New Jersey, 1974, pp. 301-307.

# HARDWARE SOFTWARE ISSUES IN MULTIMICROPROCESSOR COMPUTER ARCHITECTURES

by

C.V. Ramamoorthy, T. Krishnarao and P. Jahanian
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California at Berkeley
Berkeley, CA 94720
USA

## ABSTRACT

The advent of microprocessors as building blocks for the realization of computing devices has revolutionized the computer world. Attempts are being made to design computer systems as a network of LSI processing elements. This has been motivated by the low cost, speed, flexibility and capabilities offered by the fast-growing LSI and microprocessor technology. Such networks provide the capability of tailoring the system to a given application through functional orientation, besides providing high reliability. However, before such systems become practical there are many design issues such as control to be resolved. This paper is an attempt to focus on these issues to give a more coherent perspective. Several hardware issues and software issues related to network operation and man machine interface are discussed.

## INTRODUCTION

The advent of microprocessors as building blocks for the realization of computing devices has revolutionized the computer world. Earlier generation microprocessors were designed with specific system characteristics in mind and were mostly I/O oriented. However, as the technology evolved, they are no longer simple microprocessors of the past. A typical microprocessor, nowadays, has a powerful and flexible instruction repertoire, has a lower price tag and executes its instructions faster. Because of their speed, flexibility and powerful instruction repertoire they are becoming the building blocks for the future computer systems. These characteristics coupled with the advances in solid state memory technology have made it possible to realize a variety of computer organizations as a network of microprocessors. Such networks also provide the capability of tailoring the system to a given application such as process control, through functional orientation. As the microprocessor technology progresses, attempts are also being made to design computer systems as a distributed network of microprocessors.

### EVOLUTION OF MICROPROCESSORS

Historically Intel Corporation introduced the first microprocessor into the market. The first generation microprocessors (1970-73) were 4 bit wide LSI processors designed using p-channel MOS technology (eg. Intel 4004). The first generation microcomputer systems based on these microprocessors were originally intended for implementation as intelligent terminals. Later they were extended to 8 bit wide systems, with an 8 bit data/address I/O buses interfacing the chips with external memories. The second generation microprocessors emerged (1972) with wider lengths and n-channel technology (eg. Intel 8080, M6800, etc.). The systems based on these microprocessors could provide higher throughput because of their longer word length for both addressing and instruction. These 8 bit microprocessor architectures are very similar to minicomputers providing direct memory access channel capacity for faster input-output data transfers. Then came the third generation microprocessors that are based on bipolar and SOS technology. Architecturally these microprocessors' data widths range from 4 to 16 bits wide. Figure 1 is a simplified block

*Research supported by National Science Foundation Grant MCS72-03734-A02.

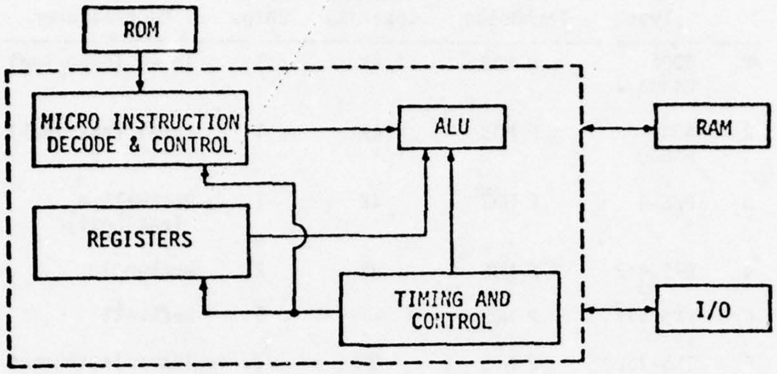


Fig. 1. Organization of a typical microprocessor.

diagram of a general microprocessor along with its memory and I/O facility. The microprocessor can communicate with the external world via this I/O facility. In addition the processor might also be able to communicate with the external world if the memory is shared by others. It is through these two facilities that a microprocessor can communicate with other microprocessors making it feasible to interconnect several microprocessors together.

Currently there are several microprocessors available in the market with varying word lengths, architectures and speeds. Tables 1a, b, c, d, e show currently available microprocessors. The trends indicate that a new generation of microprocessors with larger word lengths, higher speeds, with extensive and powerful instructions will be emerging that could encompass a wide spectrum of applications rather than be limited to terminal oriented applications. In addition, microprogrammability of these upcoming microprocessors would remove the problems associated with software thus making the use of microprocessors limited only by the imagination of the designers. One possible way of extending the use of microprocessors to applications that are so far limited by the availability of large scale computers is to connect them as a network of processors.

## DEVELOPMENT OF MICROPROCESSOR BASED COMPUTER SYSTEMS

Basically a microprocessor based network consists of a set of microprocessors connected together in a specified manner as defined by the architectural requirements. These networks might be built with the currently available microprocessors or with the LSI processors designed to perform certain specific

Table 1a. 4-bit processors.

| | Type | Technology | Address Capacity | No of Chips | Manufacturer |
|---|---|---|---|---|---|
| 1 | 4004<br>P4004 | P MOS | 4K | 1 | Intel (prog log) |
| 2 | 4040<br>P4040 | P MOS | 4K | 1 | Intel (national) |
| 3 | PPS-4 | P MOS | 4K | 1 | Rockwell (national) |
| 4 | PPS 4/2 | P MOS | 8K | 2 | Rockwell |
| 5 | PPS 4/1 | P MOS | -- | 1 | Rockwell |
| 6 | TMS-1000 | P MOS | 8K | 1 | Texas Instrument |
| 7 | SX 200 | P MOS | 1K | 1 | Essex |
| 8 | TDY-52A | P MOS | -- | 1 | Teledyne Corp. |
| 9 | IMP-4 | P MOS | 4K | -- | National |
| 10 | TCS | P MOS | 8K | 1 | National |
| 11 | MPD751D | N MOS | 4K | 1 | NEC |
| 12 | HD35404 | -- | 4K | 1 | Hitachi |
| 13 | T3271 | P MOS | 4K | 1 | Toshiba |

Table 1b. 8-bit processors.

| | Type | Technology | Address Capacity | No of Chips | Manufacturer |
|---|---|---|---|---|---|
| 1 | 8008 | P MOS | 65K | 1 | Intel (MIL) |
| 2 | 8080 | N MOS | 65K | 1 | Intel (AMD, TI, NEC, Siemens) |
| 3 | 6502 | N MOS | 65K | | MosTech (AMS) |
| 4 | 5065 | P MOS | 32K | | MosTech |
| 5 | 6800 | N MOS | 65K | 1 | Motorola (AMI) |
| 6 | F-8 | N MOS | 65K | | Fairchild (MosTech) |
| 7 | E-A 9002 | N MOS | 65K | | Electronic Arrays |
| 8 | SCAMP | P MOS | 65K | | National (Rockwell Western Digital) |
| 9 | 1801 | C MOS | 65K | | RCA |
| 10 | 1802 | C MOS | 65K | | RCA |
| 11 | COSMAC | C MOS | 65K | 2 | RCA |
| 12 | ATMAC | C MOS | 65K | 1 | RCA |
| 13 | PPS-8 | P MOS | 32K | 2 | Rockwell |
| 14 | PPS 8/2 | P MOS | 32K | 2 | Rockwell |
| 15 | 2650 | N MOS | 32K | 1 | Signetics (AMS) |
| 16 | SMS 300 | TTL-S (bipolar) | 8K | -- | Scientific Microsystems (Signetics) |
| 17 | Z-80 | N MOS | 65K | 1 | ZiLog |
| 18 | IMP-8 | P MOS | 65K | -- | National |
| 19 | CMP-8 | N MOS | 65K | -- | National |
| 20 | LP 8000 | P MOS | 65K | -- | General Instruments |
| 21 | μ Com 8 | -- | -- | -- | NEC |
| 22 | Burroughs Mini-D | P MOS | -- | 256 | Burroughs |
| 23 | MPD753D | N MOS | 64K | 1 | NEC |
| 24 | M587105 | N MOS | 64K | 1 | Mitsubishi |
| 25 | HD36401,2 | P MOS | 64K | 2 | Hitachi |
| 26 | MB8861 | -- | 64K × 8 | 1 | Fujitsu |

Table 1c. 12-bit processors.

| | Type | Technology | Address Capacity | No of Chips | Manufacturer |
|---|---|---|---|---|---|
| 1 | 6100 | C MOS | 4K | 1 | Intersil (Harris) |
| 2 | T3153<br>TLCS-12 | P MOS | 4K | 1 | Toshiba |
| 3 | T3190<br>TLCS-12A | | 96K | 1 | Toshiba |

Table 1d. 16-bit processors

| | Type | Technology | Address Capacity | No of Chips | Manufacturer |
|---|---|---|---|---|---|
| 1 | CP 1600 | N MOS | 65K | | General Instruments (EM & M) |
| 2 | MCP-1600 | N MOS | 65K | 3 | Western Digital (national) |
| 3 | IMP-16 | P MOS (bipolar to appear) | 65K | 1, 2 | National |
| 4 | PACE | P MOS | 65K | | National (Rockwell Western Digital) |
| 5 | PFL-1600A | N MOS | 128K | 3 | Panafacom |
| 6 | TMS 9900<br>SBP9900 | N MOS<br>$I^2L$ | 65K<br>-- | 1<br>-- | Texas Instruments |
| 7 | MicroPro-16 | -- | -- | -- | Plessey |
| 8 | MPD755D<br>MPD756D | N MOS | -- | 2 | NEC |
| 9 | MN1610 | N MOS | 64K × 16 | 1 | Matsushita |
| 10 | T3412 | N MOS | 64K | 1 | Toshiba |
| 11 | -- | SOS | -- | 1 | ETL |

Table 1e. Bit sliced microprocessors.

| | Type | Technology | Address Capacity | No of bits/slice | Manufacturer |
|---|---|---|---|---|---|
| 1 | 2901 | Schottky TTL | 65K | 4 | AMD (Motorola, Raytheon) |
| 2 | 9400 | TTL | 65K | | Fairchild |
| 3 | 3002 | TTL | 512 | 2 | Intel (Signetics) |
| 4 | 6701 | TTL | 65K | 4 | Monolithic Memories |
| 5 | 10800 | ECL | 65K | | Motorola |
| 6 | SBP0400 | $I^2L$ | 65K | | Texas Instruments |
| 7 | TMC 1601 | TTL-S | 32K | 4 | Transitron |
| 8 | -- | C MOS | -- | 8 | Inselek (to appear) |
| 9 | CPC/P | P MOS | 100 | 4 | National |
| 10 | RP-16 | bipolar | 65K | 4 | Raytheon |

functions. In order to exploit these advances in LSI and microprocessor technology to meet the ever increasing demand for higher throughput and availability, several studies have been made.(2,12,17,18) These studies are based on one of the two general classes of systems: the conventional multiprocessor architectures and the recently emerging distributed intelligence systems.(16) However, the success and usefulness of these networks depend on the effective means of interconnecting these microprocessors. Such interconnection mechanisms must provide a means for efficient intercommunication and control. In addition, there are problems associated with the partitioning of a job into a set of tasks and scheduling them. Also problems arise in programming the jobs on a microprocessor network and in providing an interface that makes the network organization transparent to the user. These design issues such as interconnection, and their relative importance depend on the intended goals of the systems. For example, if the microprocessor network is intended for a general purpose computing environment, a flexible but efficient interprocessor communication mechanism plays a very important role. On the other hand in a microprocessor network designed to meet specialized application requirements the communication requirements are more predictable and the design of the communication mechanisms is relatively easy. These issues related to system architecture, network control, operating systems, language support and development tools are discussed in the following sections.

## SYSTEM ORGANIZATION(1,3,5,6)

There are many different approaches to organize computer systems as a network of microprocessors. One approach is to build microprocessor networks following Flynn's classification which is based on their procedures and data streams.(6) This classification divides the systems into four categories viz. SISD (Single Instruction Stream Single Data Stream), MISD (Multiple Instruction Stream Single Data Stream), SIMD (Single Instruction and Multiple Data Stream), MIMD (Multiple Instruction Stream and Multiple Data Stream). The SISD organization is characterized by single instruction stream as in any uniprocessor system such as IBM 360, or CDC 7600. However, following these familiar large scale computer organizations a microprocessor network may be built in which each microprocessor performs a specific function. Figure 2 shows one such possible organization. In this the bipolar LSI microprocessors are configured as a set of microprocessors working in parallel. The peripheral processors and memory modules can be multiplexed through fast bus switches. The system control itself can be implemented by one or more microprocessors. However, such organization of microprocessor network requires a very complex software and the overhead incurred in scheduling the tasks, and intermodule communication is high.

The second category, MISD, is characterized by multiple instructions operating on a single data stream as in a pipeline processor such as the CDC STAR. In this type of organization each of the microprocessors can perform the function of a stage in a pipelined system. This organization can be further extended to accommodate multiple instruction streams. For example, one or more microprocessors can be organized as instruction fetching units, another set for instruction decoding and operand fetching and yet another set for execution. These sets of microprocessors when interconnected properly could execute multiple instruction streams simultaneously. Successful operation of this scheme requires schemes for tagging the instructions as they proceed along the pipe. This scheme offers high reliability because of the presence of multiple functional units. However, the main difficulty of this scheme lies with the efficient interconnection and instruction tagging schemes.

The third category, SIMD, is characterized by a single instruction operating on multiple data sets as in an array processor such as the Illiac IV. This implies that a single set of basic control sequences are applied across a number of processing units, each of which is associated with a data sequence. All processing units must act in a synchronous manner. A microprocessor network can be built based on this type of organization where the application environment requires the same computation performed on all data sets and there exists no unknown global time relationship between data sets. A slight variation of this organization is a federated system or a master/slave configuration as shown in Figure 3. A federated

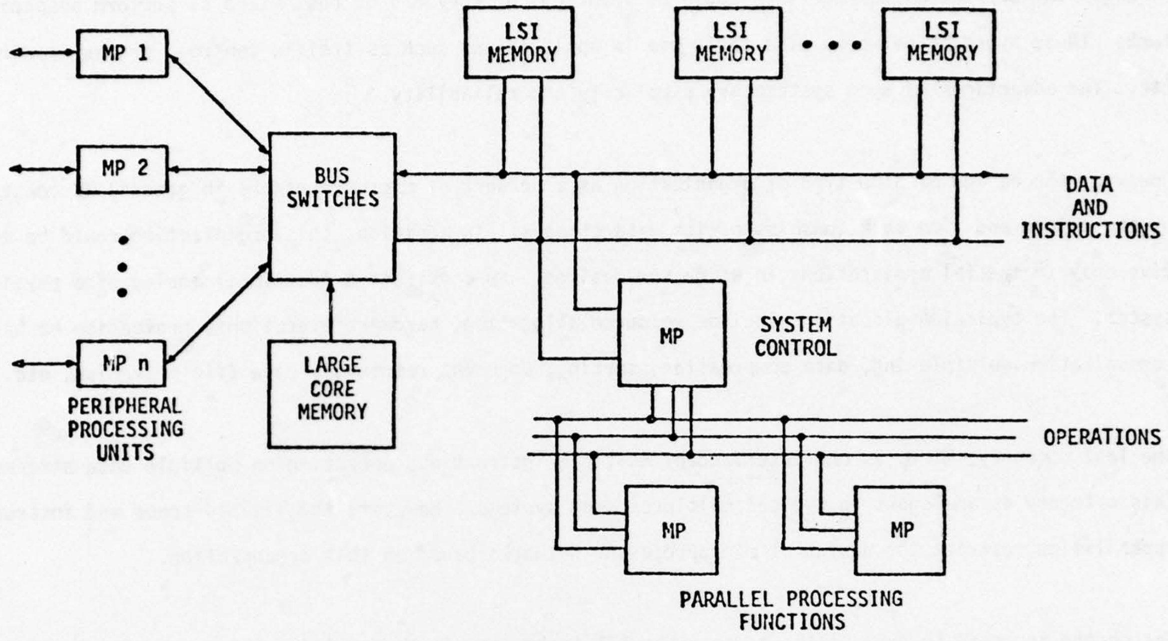


Fig. 2. A CDC type of organization.

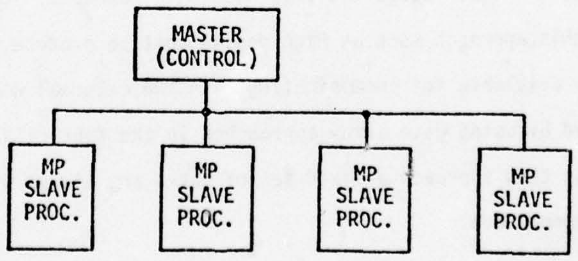


Fig. 3. Master/slave configuration.

system consists of several processors each of which is dedicated to a particular task and the tasks are performed in a multiprogrammed mode of operation. In a master slave configuration several microprocessors could be connected as slaves to a master computer which is usually a large scale computer. In these types of organizations the microprocessors could be identical or they can be customized to perform a specific task. These types of networks find their use in applications such as traffic control, process control, etc. The advantages of such systems are simplicity and reliability.

However, the design of SIMD type of organization as a network of microprocessors in general is constrained by the nature and time relationships of its data streams. In addition, this organization could be effective only in special applications in which the designer can construct a functional analog of a physical system. The typical applications include resource allocation, hardware executions, protection mechanisms, communication multiplexing, data compression, sorting, document retrieval, data file searching, etc.(21)

The last category, MIMD, is characterized by multiple instructions operating on multiple data streams. This category is analogous to typical multiprocessor systems. However, the limited speed and instruction capabilities restrict the design of microprocessor networks based on this organization.

The second approach to exploit the advances in LSI technology is to partition the processing logic and to map the partitions onto several LSI processors. Partitioning is the process of selecting portions of the system logic capable of being implemented on a single LSI chip without violating design constraints such as the number of I/O pins and number of circuits on each chip. The idea of such partitioning is to maximize the use of LSI circuits and gate to pin ratio and to minimize the total number of circuits required. Logically this approach is a mapping of each of the printed circuit boards of an SSI/MSI design onto a single LSI circuit. An example is the design of the Amdahl's V/6 machine CPU. Such partitioning and packaging had the advantage of higher speed and less non-logic hardware. But, there are certain disadvantages associated with this approach such as high design cost to produce low volume LSI circuits and the limited number of pins available for communicating with the external world. However, the initial design costs can be reduced by using gate array approaches in the fabrication phase,(3) which use two levels of metalization. In this approach a fixed set of gates are always used while providing different interconnections at the second level.

The third approach is to design computer systems as a network of functionally oriented LSI processors. These LSI processors are themselves components of a larger processor and each having an ALU unit and a small amount of memory. These processors perform various functions such as arithmetic operations, address translation, bus allocation, protection, etc. In addition, microprocessors can be used as dedicated I/O processors supporting the main processors. The I/O processor can itself be a network of microprocessors, each one performing functions such as code and format conversions, buffer and queue management, device address translation, device handling, etc. These computer systems can be further interconnected to realize multiple processor systems to achieve higher performance as shown in Figure 4.(11) However, the challenging problem in this type of organization is to find effective means of decomposing the system functions and effectively mapping onto the microprocessors while minimizing the communication overhead.

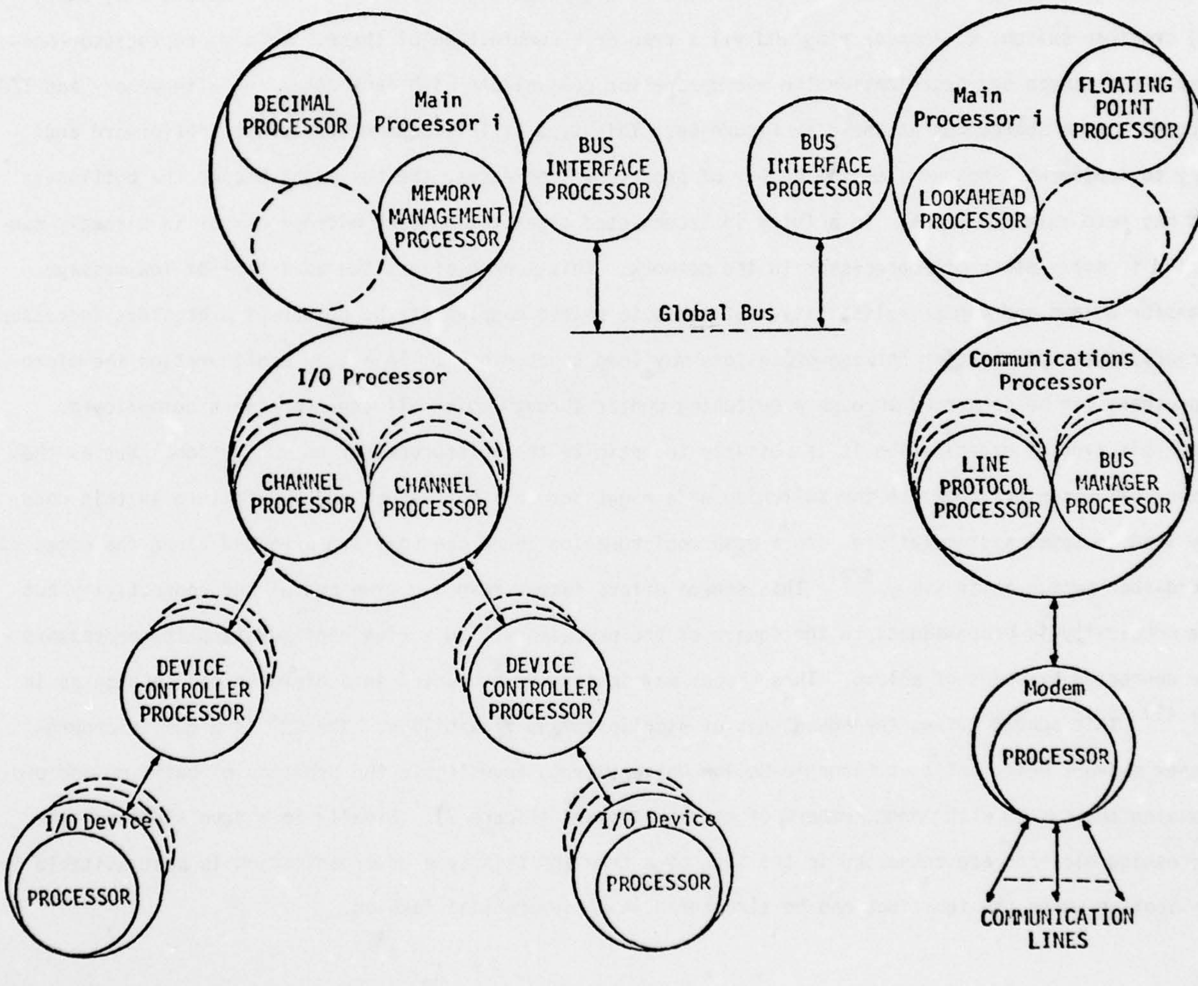


Fig. 4. Microprocessor network as a multiple processor system.(11)

The effective use of microprocessors in any one of the organizations depends on other hardware issues such as the type of interconnection or bus organization, memory organization, timing and control, addressing. These issues are further discussed in the following sections.

## BUS ORGANIZATION(20)

The interconnection scheme between the elements of any computer system is one of the most crucial factors that facilitates the transfer of data and control. This factor is more prominent in the multiple processor systems. A general configuration of a multiple microprocessor is shown in Figure 5. However, there are many different ways in which the system bus may be organized and this organization may be classified into 6 classes (Figure 6): i) time shared or common bus organization, ii) fully interconnected, iii) star, iv) crossbar switch, v) loop or ring and vi) a tree or a combination of these. In a microprocessor network based common bus organization the microprocessor communicate with each other and with memory and I/O through a time shared bus as shown in Figure 6a. This type of interconnection is straightforward and easy to implement. However, as the number of processors increases, the bus might become the bottleneck and may need multiple buses. In a fully interconnected organization each microprocessor is directly connected to every other microprocessor in the network. This scheme offers the advantage of low message transfer delays and higher reliability, but tends to become complex as the number of processors increases. Secondly the communication between processors may lead to deadlocks. In a star configuration the microprocessors can be connected through a switching center through which all the processors communicate. With this type of organization it is possible to optimize the interprocessor communication. But as the number of processors increase the switching node might become a bottleneck and any failure in this node may lead to total system failure. In a mesh configuration the processors are arranged along the edges of a crossbar switch as in C.mmp.(22) This scheme offers faster response time and higher connectivity but the complexity is prop~~ortio~~nal to the square of the processors. In a ring configuration the processors are connected by means of a loop. These loops may in turn be connected in a hierarchical fashion as in CM*.(8) This scheme offers the advantages of simplicity and flexibility. The CM* is a multimicroprocessor network being built at Carnegie-Mellon University to investigate the problems of building and programming the system with large numbers of microprocessors (Figure 7). Finally in a tree structure the processing elements are connected in the form of a tree and this type of organization is most suitable in application where the functions can be structured in a hierarchical fashion.

However, the choice of a scheme or a combination of schemes is dictated by the requirements of bandwidth, reliability, flexibility, cost, number of I/O pins, control complexity and finally the application requirements.

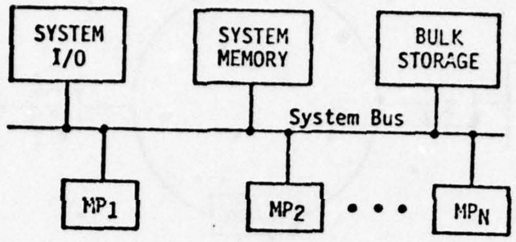


Fig. 5. General configuration of a multiple microprocessor.

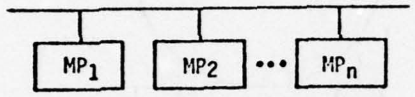


Fig. 6a. Common bus.

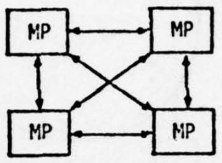


Fig. 6b. Fully interconnected.

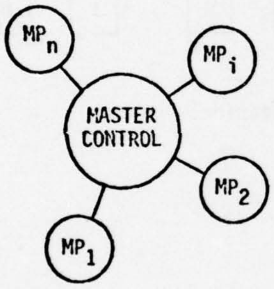


Fig. 6c. Star configuration.

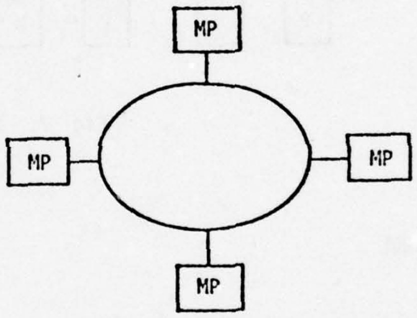


Fig. 6d. Loop configuration.

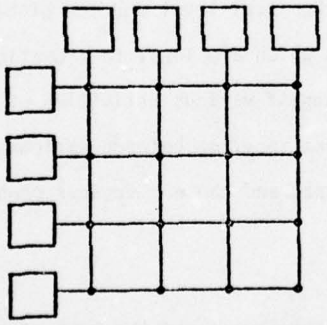


Fig. 6e. Mesh or crossbar switch.

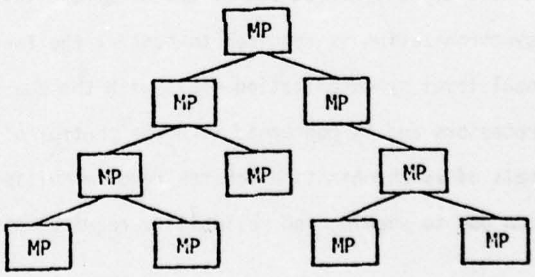


Fig. 6f. A tree structure

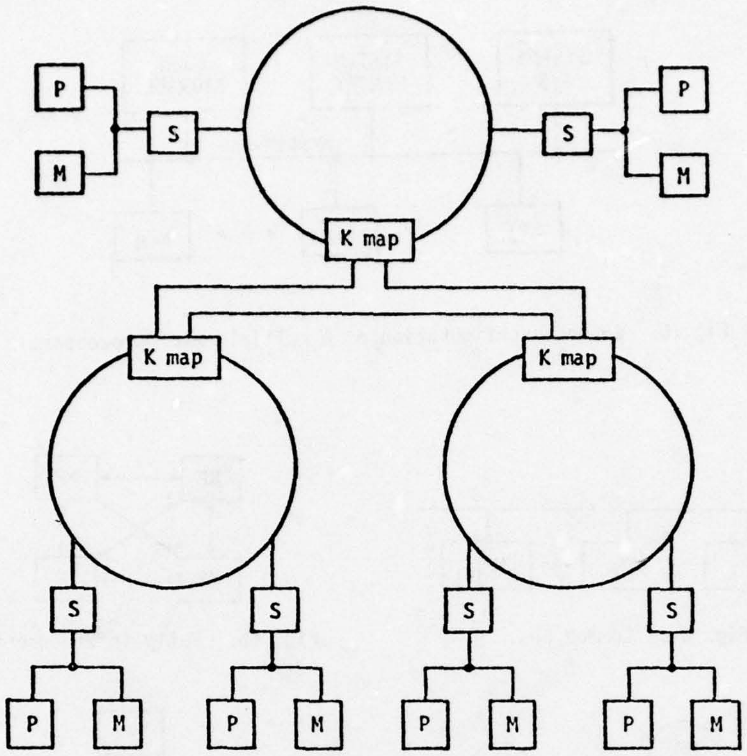


Fig. 7. CM* organization.

## TIMING AND CONTROL

The next major issue concerning the microprocessor based systems is the timing and control distribution and is mainly dependent upon the type of system organization adopted. The activities in a microprocessor network have to be synchronized at two different levels, the local level and the global level. Local level synchronization is required to control the functions which are local to a particular microprocessor. The global level synchronization deals with the coordination of various activities of the individual microprocessors and is concerned with the control of information flow between various processors. These two levels of synchronization schemes require multiple clocks and cause a serious problem in the implementation due to skewing and reliability requirements.

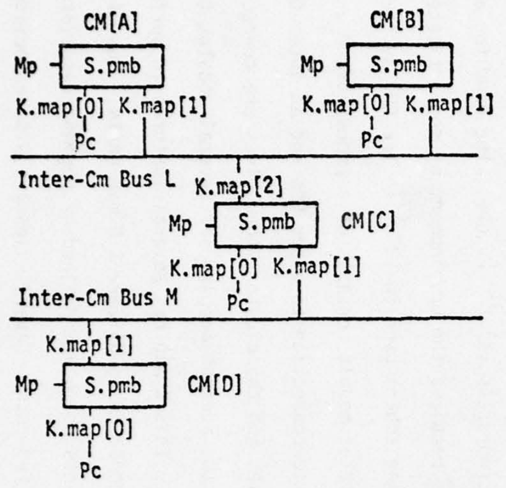
The control of the microprocessors may be centralized or distributed and the type of control is largely dictated by the system organization, nature of application and the reliability and real-time requirements. For example, in a master/slave configuration all the microprocessors may be controlled by a single master.

This type of control is relatively easy to implement but has the drawbacks of low reliability and high complexity of the master as the system grows. In a distributed control system the control functions are distributed among the processors, each performing a subset of control functions and coordinating with others in achieving the overall control of the system. Basically the control functions are implemented either in hardware or in software or using a combination of them and the nature and type of control strategy depends on the level and the scope of the individual control functions.

## ADDRESSING

The addressing technique for designating the location of data and instructions is one of the crucial factors for the success of a microprocessor network. Because of short word lengths, narrow internal buses and low speeds of microprocessors, the conventional addressing schemes often used in large scale computers are inefficient and costly. These schemes include indirect addressing, indexing, etc. In indirect addressing the address of the data address, rather than the data address itself, is included with the instruction. In indexed addressing the contents of an index register are added to the address explicitly included with the instruction. In addition there are several other addressing schemes such as direct, stack addressing that are more suitable for microprocessors.(13) However, the need for a larger address space in a multimicroprocessor based system requires the development of sophisticated schemes. One such scheme is the centralized address mapping scheme used in CM*.(7) The CM* system consists of a set of clusters of computer modules. Each computer module consists of a processor, $P_c$, a local memory, $M_p$, a number of ports, K maps which allow interconnection to other CMs and an intra-CM switch S. Each cluster consists of several of these modules and the clusters themselves are connected via inter-CM buses as shown in Figure 8a. In this system the address mapping for several modules (microprocessors) is done by a shared mapping unit called K map. This K map has several features to perform address mapping and routing of requests between modules connected to different K map units. The virtual address space of this system is divided in up to $2^{16}$ segments which are defined by segment descriptors which specify physical base address and the length of the segments. Capabilities are also associated with these segments to define protection. The K maps are capable of routing single word memory access requests independent of the processors connected. It also permits the CMs to communicate in one to many and many to one address mapping as illustrated in Figures 8b and 8c. However, this scheme is efficient in special purpose applications where a process can be bound to a processor and involves little or no multiprogramming.

Fig. 8a. A system of four computer modules.



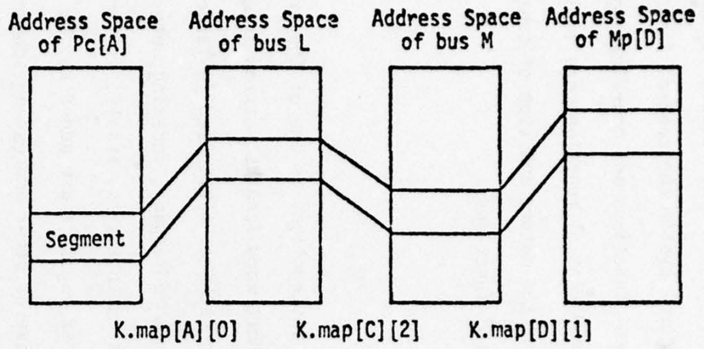


Fig. 8b. Address translation with a CM used as a switch.

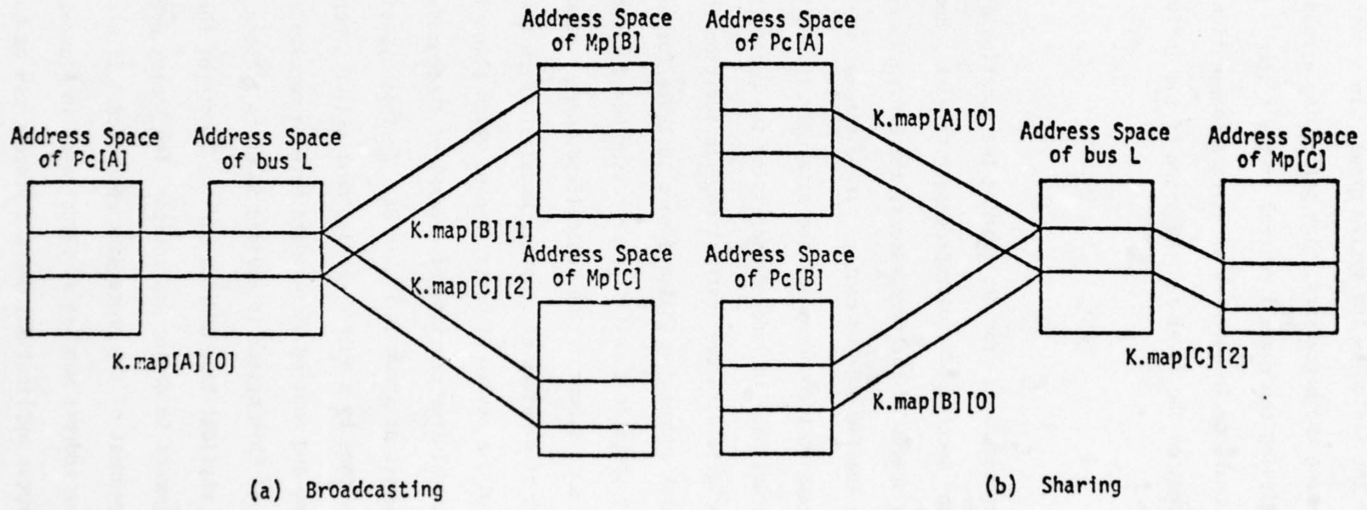


Fig. 8c. One-to-many (a) and many-to-one (b) address mapping.

## MICROPROCESSOR NETWORK SOFTWARE

Microprocessor network software may be classified into the software for the control of the network operation and the software for man-machine interface. Network operation software is related to the network operating systems and deals with the issues such as interprocessor communication task partitioning. The man-machine interface, as the name implies, refers to the means of communication between the user and the system. Users' interface with the system is through various languages, at different levels of abstraction, from bit-level to assembly level to high level. The microprocessor network user/designer interacts with the system for the purpose of system design, development and maintenance through an assortment of software tools such as compilers, simulators, debuggers, etc. These tools provide a suitable environment for the development of production software, design of new systems and maintaining the existing ones. In the following sections these software issues in a microprocessor network environment are examined.

## TASK PARTITIONING

In order to take advantage of potential advantages of the low cost LSI technology the problem of the application must be decomposed into parallel, cooperating processes. The question of whether the architecture is designed to support the application or the application is designed to suit the architecture is simply the issue of special purpose versus general purpose systems. The decomposition process or partitioning establishes the matching between the architecture and the application. Partitioning of an application refers to the division of the job into a disjoint set of processes or tasks which when integrated logically would represent the original job. Partitioning techniques similar to multitasking facilities in larger computer systems could be used. However, the limited capabilities of microprocessors limit the use of such techniques directly. These limitations arise because of limited memory and addressing capabilities associated with the microprocessors, high speed and critical time response requirements of the tasks and the communication overhead. Because of these limitations it is economical to design application oriented distributed microprocessor architectures. Such systems permit the use of firmward centered design that enables the development of systematic task partitioning.

## INTERPROCESSOR COMMUNICATION(14)

The interprocessor communication deals with the exchange of information between various microprocessors in the network. The manner in which the communication takes place between various microprocessors is primarily dictated by the system organization. However, because of the simplicity of microprocessors any techniques for interprocessor communication in microprocessor networks should be simple and uniform throughout the network. Normally the interprocessor communication is achieved through standard protocols. A protocol is a predefined set of rules that control the communication between any two or more entities such as processors. The protocol defines the status information to be exchanged and maintains coordination between various asynchronously operating processes.

The protocols in a multimicroprocessor network could be broadly divided into two classes: 1) centralized protocols and 2) distributed protocols. In a centralized system the communicating processors are local to one another and often have a common medium of communication and control such as central memory. Hence the centralized protocol often takes the form of a set of rules governing the addressing of shared memory and access to shared information. On the other hand in distributed protocols all the processes are considered remote to one another and do not deal with shared tables and memory but deal with explicit messages between any two communicating processes. Because of this distributed nature and lack of any single control over the entire environment the problems of addressing, routing, flow control and the error control become extremely important. However, the choice between centralized and distributed protocols is largely dictated by the size and the nature of the application, the transmission rates, delays, reliability and the flexibility requirements.

## LANGUAGE ISSUES

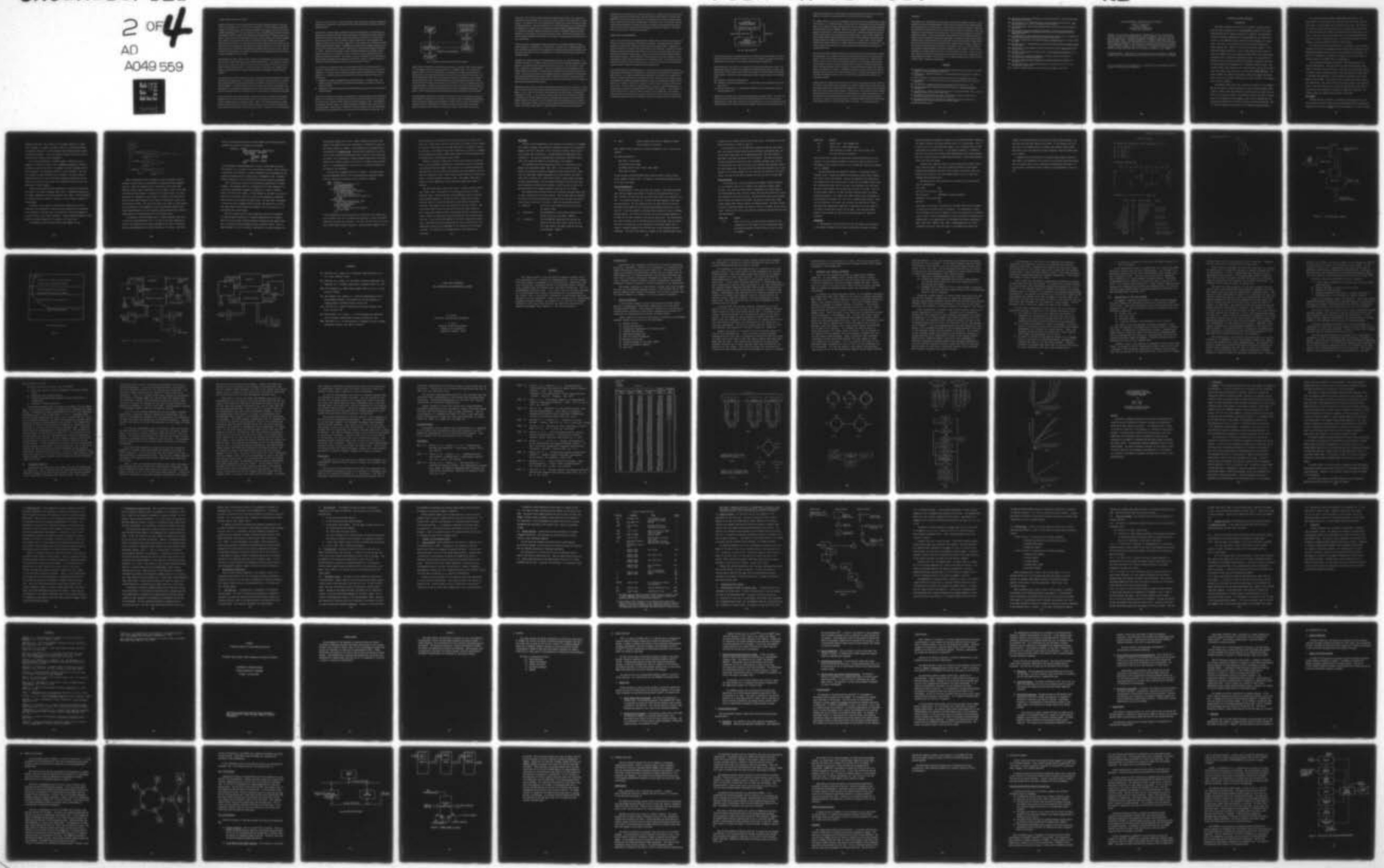
Microcomputer languages fall in the same classes as minicomputer languages: machine, assembly and high-level. However, the relative frequency of use, the number of languages in each class and the conditions under which the languages are used are quite different. For example, there is a great deal of application work being done directly in machine language. This is mainly because of the "primitive" nature of the atmosphere in which much of the work is being done. The configurations of most microcomputer systems are restricted to "bread board" systems that lack the memory and input/output capacity to support program development in assembly language. Many of the programs for such systems are written in octal or hexadecimal, directly onto PROMs, using inexpensive PROM programming equipment.

Recent trends in microprocessor software have been toward development of tools which make programming of these systems easier and less error prone. For example, perfectly useful assembly language programming tools are available for the most popular microcomputers. These include text editing programs similar to those currently available on minicomputers, assemblers, loaders and on-line debugging aids that make it possible to set up microcomputer program development environments that are comparable to minicomputer

AD-A049 559 COLORADO STATE UNIV FORT COLLINS DEPT OF ELECTRICAL --ETC F/G 9/2
PROCEEDINGS OF THE ANNUAL ROCKY MOUNTAIN SYMPOSIUM ON MICROCOMP--ETC(U)
SEP 77 M ANDREWS, S MCCORMICK AFOSR-77-3367

UNCLASSIFIED AFOSR-TR-78-0017 NL

2 OF 4
AD
A049 559



OF 4

9 559

assembly language development environments.

Progress is being made in the area of system level and high level language development, but the repertoire of these languages is not as rich as it is for minicomputers. The most commonly used high-level language for microprocessors is the PL/M language developed by Intel Corporation for use with the 8008 and 8080 microprocessors.(10) Other derivatives of PL/M such as PL/M6800(9) by Motorola and the National Semiconductor's PL/M(4) have recently been developed for the microprocessors(4) produced by these companies. There is interest in the development of compilers for other languages such as BASIC and FORTRAN but it remains to be seen whether this interest is sufficiently intense to promote actual commercial development of these compilers. The small word size of the most common microprocessors has been an impediment to their use in applications that require heavy calculating capabilities. As a result, there has been less motivation for FORTRAN and BASIC compilers than might otherwise have been the case. With the advent of 16-bit microcomputers, this situation is changing and will probably promote an increased level of compiler development in the near future.

Microprocessor based computer networks utilize a collection of microprocessors as their basic functional unit to achieve a higher computational power, reliability and flexibility. It was mentioned in previous sections that such networks may replace not only the traditional I/O controllers, but also the minicomputers and even more powerful machines. As such, such factors as small memory size, limited I/O capabilities and operating systems are no longer a stumbling block to development of suitable high-level languages.

A microprocessor based network consisting of a number of possibly nonhomogeneous microprocessors provides new challenges to language designers. Obviously some of the emphasis on language features are shifted as the system characteristics and application requirements are different from those of a single processor based microcomputer. Microcomputer networks have usually larger storage capacity, provide better computational power, a more powerful I/O capability along with a higher degree of interface complexity. Thus, they favor the usage of higher level languages than what is now available for conventional microcomputers.

The most common means of man-machine interface in microprocessor networks is through assembly language programming. Due to a larger storage generally available in a network environment, it is possible to reside the assembler directly in the computer system. However, if the processing elements are nonhomogeneous, there will be the need for several resident assemblers, each for a particular microprocessor, thus causing a serious drainage on the amount of storage available. Therefore, for a multi-microprocessor network it is desirable to have a single meta assembler, capable of supporting a number of nonhomogeneous microprocessors, thus saving in storage without sacrificing the flexibility and reliability offered in a network environment.

Meta assembler is a general purpose assembler which accepts the characteristics of a particular microprocessor as input and then translates a machine independent, but microprocessor oriented, assembly language into the equivalent machine code for the target microprocessor (Figure 9). The machine characteristics may be input as a series of subroutines, tables, instructions of a suitable specification language or a mixture of any of the three. There are few, if any, proposals for a suitable meta assembler and the supporting assembly language. Davidson(23) proposed a microprocessor programming language which was similar to PDP-11 assembly code in format and contained most of the existing microcomputer instructions as a subset. Instead of a meta assembler, however, he proposed development of an assembler for each popular microprocessor, with translation to be done in four passes using a combination of macro-like expansion and compilation techniques. This approach is similar to the proposal for an UNCOL (stands for UNiversal COmputer Language) which was proposed in the late 50's as a means of translator writing for a large class of computers, although in this case, the scope is much more limited.

It is felt however that Davidson's scheme in writing separate assemblers for each microprocessor is restrictive. Many functions such as bit packing, resolving of forward references, etc. are essentially independent of a particular processor architecture and may be uniformly implemented for all the microprocessors. It is therefore proposed that the construction of the Meta Assembler be broken into two distinct parts:

a) Development of processor independent modules of the Meta Assembler in a 'skeletal' form. Such modules are machine independent but are controlled by the parametric inputs which constitute the machine description.

b) Development of the medium through which the processor dependent characteristics can be described to the Meta Assembler.

Thus there are two phases in construction of an assembler for a processing element of the microprocessor network. In the first phase all the general purpose modules of the Meta Assembler which are necessary for the translation of the Meta Assembly language into machine language are written. These modules may be developed on a powerful computer in a suitable language such as FORTRAN or BASIC, thus called the Cross Meta Assembler. Once this phase is completed, it need not be repeated for any new microprocessor. The second phase involves preparation of the characteristics of the individual microprocessors. The

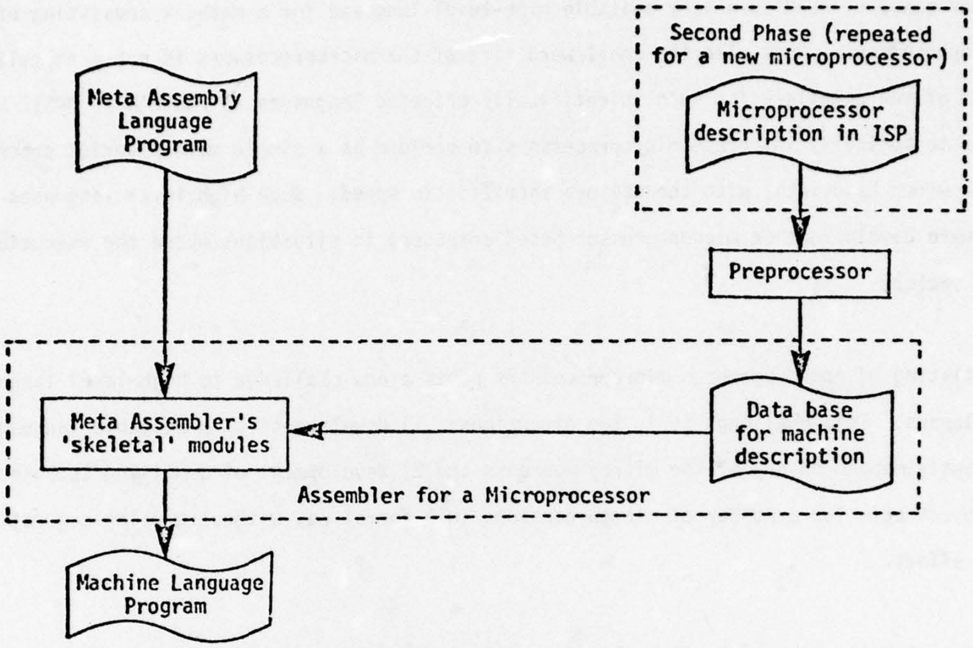

Second Phase (repeated for a new microprocessor)
Microprocessor description in ISP
Preprocessor
Meta Assembly Language Program
Meta Assembler's 'skeletal' modules
Data base for machine description
Assembler for a Microprocessor
Machine Language Program

Fig. 9. Generation of an assembler using the Meta Assembler.

medium in which this description is provided may vary and may be, among other things, the host language, i.e. the language in which the first phase was written, e.g. FORTRAN or BASIC. It is felt, however, because of the nature of the description which is oriented toward the hardware characteristics of the microprocessors a conventional hardware description such as ISP is more suitable for this purpose. The ISP has the capability which allows the microprocessor's operational characteristics such as assembly instruction repertoire, machine language instructions and their representative bit patterns, register transfer paths, interrupt flags, error conditions to be described in a precise and unambiguous manner and yet be manipulated by an automatic processor. While the first phase in the design of the Meta Assembler is carried on only once, the second phase should be repeated for each nonhomogeneous microprocessor in the network. Figure 9 illustrates the different phases and the operational diagram of the Meta Assembler.

High level language issues related to microprocessor networks are similar to the Assembly language and the development of assemblers for such a system. If the network consists of homogeneous microprocessors, the task of high level language compiler development is greatly simplified. Again, the enhancement in computational power of the computer due to harmonious activities of the individual microprocessor elements, coupled with a wider variety of I/O devices and a larger storage capacity bring about an environment in which high-level language compilers can conveniently be developed. Such a compiler is however totally machine dependent in order to achieve a high utilization of the target machine's critical resources. For example, PL/M is a very suitable high-level language for a network consisting of individual 8080 microprocessors. Note also the small word size of the microprocessors is not a stumbling block to development of the compilers for such scientifically oriented languages as FORTRAN or BASIC because it is possible to logically integrate microprocessors to perform as a single more powerful processing element with a wider bandwidth, with the obvious sacrifice in speed. Such high level languages may be used for software development on microprocessor based computers in situations where the execution time is not a crucial factor.

A network consisting of nonhomogeneous microprocessors poses a new challenge to high-level language compiler developers. This challenge is in two directions: 1) development of a suitable language which can be efficiently mapped on any of the microprocessors and 2) development of compiling techniques which can produce object code for a number of microprocessors in a formal way without causing a great deal of reprogramming effort.

Development of a high-level language for multi-microprocessor networks is an open-ended problem. Most present-day languages are not suitable candidates since they are either too machine independent or machine dependent. Any proposed high-level language should consist of two parts. The first part is a kernel which is common among all microprocessors. This core of the language is machine independent and its features reflect the requirements of the application environment and suit the algorithms which are to be written. The second part is, however, machine dependent and varies from microprocessor to microprocessor. While the machine dependent features provide a means of controlling the network's resources and their efficient utilization, such features may be put aside in many cases where the network is to be viewed as a general purpose microprocessor-based computer.

Generation of efficient code for a number of nonhomogeneous microprocessors is not an easy task. In practice it is required that for each microprocessor new code generation modules must be written and used when needed. This is a tedious task and incurs a major programming cost when the number of nonhomogeneous microprocessors is large. In very recent years, there has been some effort to alleviate this problem. In particular, there have been attempts to apply the techniques already developed for transporting the compilers between various machines in order to develop automatic code generating systems for microprocessors. Bunza proposes a scheme, involving a system to generate machine code for many different microprocessors, utilizing a single compiler, a single code generator and processor-specific data bases.

Adaptation of the code generator to new processor architectures is claimed to be limited to the creation of a single processor description data base. The code generating system translates or interprets BCPL OCODE(19) with a hierarchical operator definition tree, manipulates data classes, and generates machine code utilizing a processor description language which is a derivative of ISP. This and other techniques require a thorough knowledge of the processor description language and the intermediate medium (BCPL in this case) and are subject to manual tuning after the processor description is complete. Their usefulness is yet to be seen in a real commercial environment.

## SOFTWARE TOOLS FOR SYSTEM DEVELOPMENT

Another area of major interest in microprocessor-based computer systems is the software aids which can be utilized for the design, development and evaluation of new systems. There have already been significant innovations in this direction, especially in single microprocessor computer systems. Simulators have long been introduced for the design, checkout and evaluation of microprocessors. Simulators are software packages which execute the operations specified by an input medium (e.g. program process signals, etc.) and simulate the response of target systems. Simulators are convenient tools for verifying the design before actual construction. Most manufacturers of microprocessors provide 'canned' simulators for their microprocessors, usually written in FORTRAN and run on a time-sharing system. In very recent years there have been some attempts to deviate from the concept of 'canned' simulator packages and to develop tools which can be used to 'create' new simulators for various microcomputers.

SIMIC (stands for SImulator for MICroprocessors) is a microcomputer software development and evaluation package which has been designed and implemented on a Digital Equipment Corporation PDP-11 minicomputer.(15) SIMIC consists of a main module containing machine independent routines and data and a submodule which contains a particular microprocessor characteristic (Figure 10). Generation of a new simulator requires the preparation of microprocessor dependent information in tables and subroutines which organize the submodule part of the SIMIC. Programming of a submodule is oriented to those structures in which microprocessors differ. As part of the SIMIC, a general purpose set of programs, called the MICRO module, has been developed to facilitate the implementation of microcomputer cross-assemblers. A cross-assembler for a specific microprocessor is formed by linking a submodule to the main MICRO module. This submodule contains tables and a specialized subroutine which tailors the main module to a specific microprocessor. The assembler implemented with MICRO is applicable to generate microcomputer software for control tasks.

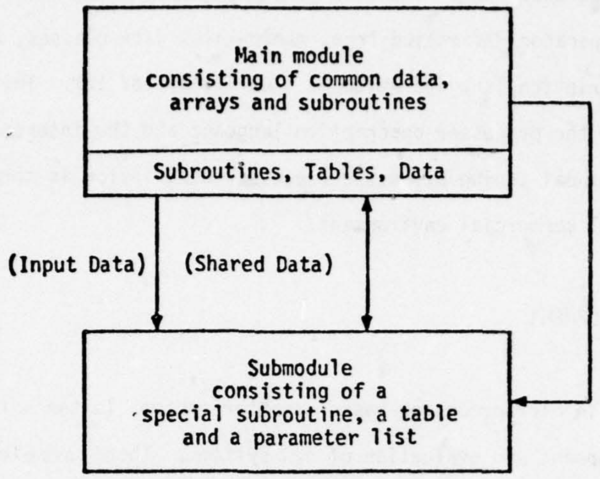

Main module consisting of common data, arrays and subroutines
Subroutines, Tables, Data
(Input Data)
(Shared Data)
(Execution)
Submodule consisting of a special subroutine, a table and a parameter list

Fig. 10. SIMIC modules.(15)

The SIMIC package provides the bookkeeping mechanism for evaluating microprocessor software in terms of design effort and speed of operation and hardware in terms of the computer and external hardware costs. The evaluation process focuses on the tradeoffs between the design and computer costs.

Simulator generators such as SIMIC are very desirable for microprocessor-based computer systems. In a reconfigurable network environment where the application requirements demand changes in system characteristics, the 'canned' simulators cannot respond to such demands in sufficiently short time to justify their usefulness. Simulation of a multi-microprocessor network, in general, consists of the following steps:

(a) simulation of individual processing elements

(b) simulation of the physical configuration, i.e. processor-memory module interconnections, bus structure, I/O, etc.

(c) simulation of the control, i.e. interprocessor communication, task distribution and allocation, priority assignment, etc.

Simulation is an important step in the design and implementation of computer networks. It not only specifies the characteristics that individual processing elements should have to meet the application requirements (e.g., speed, proper instruction set, proper bandwidth, etc.) but it also suggests the best network configuration, bus structure and any other parameters which influence the design constraints. In many cases, simulation runs of a network detect run-time hazardous conditions which might arise such as deadlock, race condition, integrity problems, prior to the construction of the network.

If a network simulator is to be used as a development tool to answer some of the problems above, it should not be canned, i.e., it should not simulate a specific network of fixed characteristics and configuration. What is in fact needed is a software system development tool to be called INTERactive SImulator Generator for microprocessor networks or INTERSIG. INTERSIG can be viewed as a general purpose network simulation package to be available on a time-sharing system which provides a simulation environment for the design checkout and maintenance of the microprocessor based computer network. Similar to SIMIC, INTERSIG consists of general machine independent modules which interface with the outside world through tables automatically created by a processor from the network description. It is felt, however, that the network characteristics should be described via a suitable specification language such as an augmented ISP. This allows for interactive development of the network description and early checkout of the design using standard software tools.

Through the specification language instructions, the user specifies the rigid characteristics of individual microprocessors, e.g. instruction set and the equivalent machine code, word size, addressing modes, registers, inter and intra submodule communication paths, timing requirements, etc. Such description should in fact be sufficient to simulate the complete operations of the individual processing elements. Also instructions should be provided to describe the network configuration, including interconnection between the processing elements and other subsystems, bus structure and other characteristics which are crucial in the definition of a network. The complete specification 'program' is input to the INTERSIG which then processes them just like any other translator, fills up its internal data structure with the information and subsequently tailors itself to the specification, producing a new simulator for the microprocessor network.

INTERSIG resides on a powerful time-sharing system to utilize the run-time supports normally available on such systems. General purpose simulation packages such as SIMIC or INTERSIG become more important as the microprocessors become more and more common as building blocks for the design and construction of powerful application-oriented or general purpose computers. Along with them, a full complement of system software production aids such as cross-meta compilers, cross-meta assemblers are needed so that appropriate software processors can be developed and checked out on the simulators even before the computer system is constructed. In this way, a very realistic measure of all system parameters and their cost-effectiveness can be obtained very early in the design phases before a major budget is committed to the project.

## CONCLUSIONS

The revolution of the microprocessor at low cost shows a new trend in the computer architecture. As the microprocessors become the building blocks for computer systems, the gap between the component designer and the system designer is becoming narrower. However, before a microprocessor network as a general purpose system becomes a reality, there are several other issues that need to be investigated. First, investigation on the architecture for future microprocessors: The current day microprocessors are limited in capabilities and power. In order to overcome these limitations the future microprocessors should have features to provide flexibility and performance improvements. These features include powerful communication capabilities to include protocols such as SDLC, higher data processing bandwidth, language support features, OP code flexibility and larger addressing range. Secondly, tools to describe, simulate, emulate and design a multi-microcomputer system need to be developed. Tools such as SIMIC or INTERSIG as general purpose simulation packages provide powerful capabilities in this direction. Along with them, a complete set of software production aids such as cross-meta compilers are needed. Finally very little work is done on the testing side, and to circumvent the difficulty of testing tools integrated with the system development are needed. All these issues are to be studied thoroughly before a major commitment is made to design a microprocessor based system.

## REFERENCES

(1) R.G. Arnold et. al., "A Hierarchical, Restructurable, Multimicroprocessor Architecture," 3rd Annual Symposium on Computer Architecture, January 1976.

(2) A. Baum et. al., "Hardware Considerations in a Microcomputer Multiprocessing System," COMPCON 75, February 1975.

(3) Barry R. Borgerson, "The Viability of Multimicroprocessor Systems," Computer, January 1976.

(4) D.A. Cassell et. al., "The Current State of the Art in Microprocessor Software," ACM Conference, October 1975.

(5) P.H. Enslow (ed.), Multiprocessors and Parallel Processing, Wiley Publishing Co., 1974.

(6) M.J. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Trans. on Computers, September 1972.

(7) S.H. Fuller et. al., "Computer Modules--An Architecture for Large Digital Modules," Proc. 1st Annual Symposium on Computer Architecture, December 1973.

(8) S.H. Fuller, D.P. Siewiorek, and R.J. Swan, "The Design of the a Multimicrocomputer System," 3rd Annual Symposium on Computer Architecture, January 1976.

(9) D.H. Fylstra, "PL/M6800. A Compatible High Level Language for the Motorola Microprocessor," Symposium on Trends and Applications 1976, Micro and Mini Systems.

(10) Intel PL/M Reference Manual

(11) E.D. Jensen, "The Influence of Microprocessors on Computer Architecture: Distributed Processing," ACM Conference, October 1975.

(12) J.E. Juliussen et. al., "Multiple Microprocessors with Common Main and Control Memories," IEEE Trans. on Computers, Vol. C-22, No. 11, November 1973, pp. 999-1007.

(13) L.A. Leventhal and W.C. Walsh, "Addressing Considerations in Microprocessor Design," COMPCON 77, February 1977.

(14) R.M. Metcalfe, "Strategies for Interprocess Communication in a Distributed Computing System," Proc. Symposium on Computer Communications Network and Teletraffic, Polytechnic Institute of Brooklyn, April 1972.

(15) C.P. Neumann et. al., "SIMIC--A Microcomputer Development and Evaluation Tool," Proc. Symposium on Trends and Applications 1976, Micro and Mini Systems, IEEE Press, 1976.

(16) H.A. Raphael, "Joining Micros into Intelligent Networks," Electronic Design, Vol. 5, March 1, 1975, pp. 52-57.

(17) V.K. Ravindrau et. al., "Characterization of Multiple Microprocessing Networks," COMPCON 73 Digest, 1973, pp. 133-137.

(18) G. Reyling, "Performance and Control of Multiple Microprocessor Systems," Computer Design, March 1974, pp. 81-87.

(19) M. Richards, "The Portability of the BCPL Compiler," Software Practice and Experiences, Vol. 1, 1971, pp. 135-146.

(20) K.J. Thurber et. al., "A Systematic Approach to the Design of Digital Busing Structures," Fall Joint Computer Conference, December 1972.

(21) K.J. Thurber, Large Scale Computer Architecture. Parallel and Associative Processors, Hay Publishers, 1976.

(22) W.A. Wulf and C.G. Bell, "C.mmp--A Multimicroprocessor," AFIPS Conference Proc., Vol. 41, F .l Joint Computer Conference, 1972.

(23) T. Opdenyk, "Software Considerations for Microprocessors," Computer, January 1976.

# A MICRO-PROGRAMMED SCANNING AND PARSING PROCESSOR

Stephen A. Skedzeleski
Bruce R. Rowland
Computer Sciences Department
University of Wisconsin

ABSTRACT: We describe a general-purpose, asynchronous, lexical and syntactic processor for use in the compilation of computer programs. It is table-driven, with the capability of translating different languages by loading the processor with different tables. Automatic methods (i.e., computer algorithms) exist to generate these tables, and these algorithms are practical both in time and space. The processor is designed to use bipolar bit-sliced microprocessors, and consists of two independent units; a lexical analyzer (scanner) and a syntactic analyzer (parser), which communicate via a shared latch. We argue that the time and space benefits of such a processor make it an attractive addition to any computer.

DESCRIPTIVE TERMS: deterministic finite automaton, lexical analysis, LR parsing, microprocessor, push-down automaton, syntactic analysis, table-driven techniques

The first author has been supported by a fellowship from the Mathematics Research Center of the University of Wisconsin.

## SCANNING AND PARSING PROCESSOR

### INTRODUCTION

This paper describes the front end of a compiler (lexical scan and syntactic parse) implemented as a microprogrammed processor to be placed between memory and the central processor of a computer. The generalized scanning and parsing algorithms are microprogrammed and are table-driven by data in local RAMS that may be reloaded for each language to be translated. The input to the pre-processor is the character stream of the source program code. Output is in the form of two data streams: 1) the production numbers realized in a left to right bottom-up parse (used to drive the compiler) and 2) the character strings that are the tokens being recognized (used in subsequent compilation steps). The pre-processor is organized into two distinct processors that execute in parallel with their own local memories and communicate via a dedicated interrupt system. The scanner delimits input characters into token character strings and delivers a token code number to the parser via a shared buffer (TOKEN REGISTER). The parser consumes the tokens and identifies context-free grammar reductions which are sent to a compiler executing in the main processor. This organization is shown in figure 1.

The compiler writer needs only to specify the syntax of the language in BNF and the tokens as regular expressions. We have written computer algorithms to convert these specifications to tables, which are loaded into the RAMS of the scanner and parser. The compiler writer is then able to concentrate on the semantics of the language, secure in the knowledge that the scanning and parsing will be handled in exactly the manner specified. This is a distinct advantage over using ad hoc scanning and parsing methods.

The scanner and parser operate asynchronously in parallel. One shared buffer is placed between the units into which the scanner deposits token codes or error messages later consumed by the parser. The latch buffer contains an interrupt output line that is set high by the parser when it clears the buffer and is pulled low again when the scanner writes a new token. Both units latch the interrupt line in their micro-sequencers and must test it prior to any buffer usage. The character string out from the scanner to the main processor follows the same protocol.

The scanner delivers two special codes to the parser. Zero indicates a scanning error (this is determined from the tables) and 255 marks the end of transmission (meaning that the source code end marker has been read). At the end of transmission, the scanner and parser go into an idle loop and require a pulse on the START line to be reactivated.

The processor was designed using Intel's 3000 series bipolar, bit-sliced microprocessors. They were chosen for their high speed and large number of processor inputs and outputs. Each part of the processor consists of an 8-bit central processing element (CPE) constructed from 2-bit slices, a look-ahead carry generator, a micro instruction sequencer, 512 x 8-bit ROMS, and 4K bit RAMS. The processors share an Intel 3212 latch for internal communication, and use more 3212's for buffering output to the main processor. The 3212 latches are well suited for this purpose, since they can be polled by either producer or consumer for empty/full conditions. The scanner accepts the program stream through a FIFO buffer.

## THE SCANNER

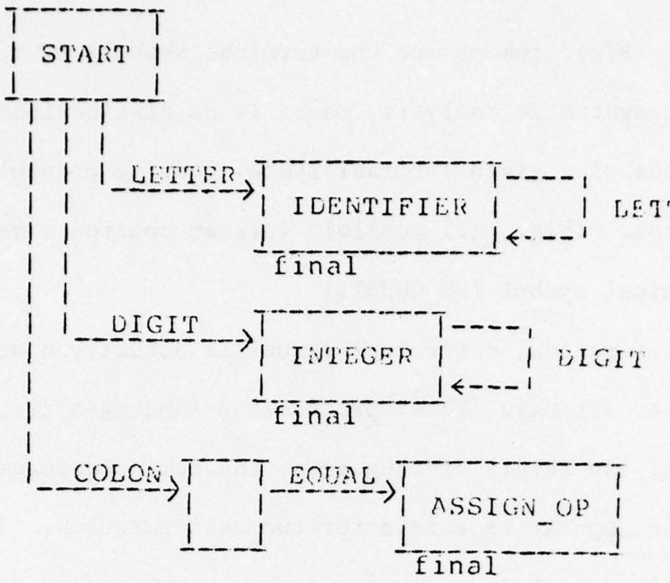
Lexical analysis in compiler or translator terminology refers to the partitioning of an input string into non-overlapping character strings that represent the lexicons or tokens of the source language. In terms of formal language theory (AU, 1972), tokens are the terminal symbols of a formal syntax language. In syntactic analysis, there is no distinction between two separate instances of a given terminal symbol despite possible character string differences. (E.g., all possible integer constants may be represented by the terminal symbol INT_CONST.)

Due to these differences, a terminal symbol is actually a set, or language, of character strings. Thus, programming languages are normally specified in terms of two levels of languages, the token languages and the syntax language. The separation exists for two main purposes. The first is to provide a simpler vocabulary for the syntax. The second and more important reason is to allow two distinct recognition methods for the two levels, each tailored to best handle (via appropriate space and time optimizations) its input language.

Tokens, which generally include identifiers, literals, and delimiters, are assumed to be regular languages (GRE, 1971). A recognizer designed just for regular languages is the finite automaton (HU, 1969). A finite automaton is a machine that moves through a sequence of states (finite in number) in a state graph.

Transitions in the graph are directed arcs labelled with a set of characters, the recognition of which allows the transition to be performed. If the automaton is deterministic, each character in the input alphabet is allowed to select only a single transition. Certain states are marked as final states that signal recognition of a member of the regular set.

A deterministic finite automaton state graph example follows:



When a deterministic finite automaton is used for scanning a source program, the final states of the machine are labelled with the recognized token set. When the automaton enters a final state and the next input character does not label a transition out of the state, a code for the token is emitted and the automaton is restarted in its initial state. The automaton acts as a transducer in this respect. If there is any other point at which the next input character does not select a transition, an error code can be emitted instead. The string that makes up the token is often required for later analysis. The automaton can have its transitions labelled with flags denoting whether or not the character causing the transition should be concatenated to the string being built as the current token.

To implement a finite automaton model, a simple algorithm exists which is driven by a two-dimensional table. The rows of the table represent the states of the machine and the columns correspond to the sets of characters that are distinguished by the lexical analysis unit or scanner. Each table entry can be characterized by the following PASCAL record which denotes the possible next actions to be taken by the scanner:

```
TableEntry : RECORD
               CharacterDisposition : (Emit,Forget);
               CASE FinalState : BOOLEAN OF
                 TRUE: RECORD
                         TokenCode : INTEGER
                         LookAhead : BOOLEAN
                         END
                 FALSE: NextState : INTEGER
               END;
```

In the TableEntry, CharacterDisposition is used to decide when the transition character is to be added to the token string or deleted. When a state is a final state, the TokenCode is emitted separately from the character and the transition character must be scanned again if LookAhead is true. Otherwise another transition is to be followed and the new state entered is NextState.

Such a table-driven scanner can be efficiently implemented in flexible firmware. The scanning tables for a particular set of token languages (for a particular programming language) are loaded into a series of 4096-bit RAM's of 64 rows and 64 columns. In this design, ten such RAM's are used and eight bits of each table entry can be used for the NextState or TokenCode. Two bits suffice for encoding the other fields. One input port is necessary to accept the source language character string. One output port is used to emit token codes to the parsing unit; another to emit the characters to be concatenated into token strings.

There is a special case in token recognition that must be considered for time and space efficiency in a firmware implementation of a scanner. There often occur several dedicated character strings that are formally members of a particular token language but used as distinct tokens. The most common example is the use of selected "identifiers" to denote delimiters for which special symbols do not exist. BEGIN, INTEGER, and END are three examples from PASCAL that fit the description of "identifier" except for their being reserved as symbols. Such exceptions can be grouped under a classification of reserved words. Reserved word recognition can be built into the table structure described but tends to make the table sparse and very large. As an alternative, a pattern matching algorithm can be formulated and the reserved strings can be stored in a tightly packed representation. In the implementation, several rows of the RAM are dedicated to the reserved word encodings.

The scan unit is organized as shown in figure 3. The ROM contains the microprogrammed version of the following general algorithm in which input and output and a procedure call are replaced by comments.

```
     START : STATE:=C;
     LOOP :  (* fetch input character C *);
             CN:=CHARACTERSETNUM|C|;
             ENTRY:=TABLE|STATE,CN|;
          IF ENTRY.FinalState then begin
             (* send ENTRY.TokenCode to parser *);
             if not ENTRY.LookAhead
                then (* send character C *)
                else (* reset scan pointer to reuse C *);
             goto START
          end else begin
             if ENTRY.CharacterDisposition = Emit
                then (* send character C *);
             STATE:=ENTRY.NextState;
             if STATE > TABLEND
                then (* start reserved word search *)
                else goto LOOP
             end;
```

In the implementation, the high order bits of each data word (either table entry, character set code, or reserved word character or mark) are directly routed into the 3001 micro-sequencer. The other data bits enter one of the four two-bit-sliced processor registers. They are either compared to the input character (in the reserved word match), buffered to be sent to the parser (as a token code), used as a row selector in the RAM (as a next state), or used as a column selector in the RAM (when data is a character or character set code). The character input from a FIFO buffer is saved in a register of the processor and used as a RAM column selector to find the character set and perhaps sent to the character output stream.

Whenever a character, token code, or error code is sent to an external device or the asynchronous parsing unit, the scanner must poll the interrupt line associated with the shared latch to see if it is currently holding unprocessed data. For this reason, the interrupt lines feed directly into the 3001 micro-sequencer as well. When an input stream character is finally consumed, a new input character request is sent. The scanner loops while it awaits an initial START pulse or the latch-ready signal when output is buffered.

The microinstructions contain five fields: function, carry-in, mask, latch select, jump. The function is the instruction code of the 3002 processor slices. Carry-in sets the least significant carry for the 3003 carry lookahead generator. The mask is used in various ways to enable selective views of the data bits and give a wider variety of processor functions. This aspect of the Intel processor proved very fruitful. The latch select lines are used to control the input and output latches as well as the memory address latches, since busses are used to overlap line usage. Finally, the jump field is the 3001 sequencer next address instruction code. Four 512-word 8-bit ROM units are used to contain the instructions. Approximately 25% of the ROM words are used as instructions. The compactness allows room for enhancement in the capability of the scanner if needed. The utilization of the RAM depends on the language to be translated.

## THE PARSER

Parsing is the determination of the structure of an element of a language. For computer languages, the structure is specified in terms of a content-free grammer, such as by using BNF. In the following discussion, LHS(l) denotes the left-hand side of production 1, and RHS denotes the right-hand side of production 1. The parser recognizes when a production in the BNF is applicable, and returns the number of the production to the compiler.

The algorithm used in this parsing processor is a modification of LR parsing first described by Knuth (KNU, 1965). LR is a powerful parsing method, but building the tables is expensive in both time and space. Therefore, the tables most commonly generated are not LR(1) tables, but SLR(1) (DER, 1971) or LALR(1) (AJ , 1974) tables instead. These methods are based on LR(0) parsing methods, with less precise look-ahead sets than LR(1) uses. Tables generated by LR(0), SLR(1), LALR(1) or LR(1) can each be used by the parsing processor; the parsing algorithm is the same for all of them.

The parsing tables describe actions for a push-down automaton whose stack is used to keep track of the states visited. When an input token is received, the action for the (state,token) pair is looked up and executed (the current state is kept on the top of the stack). There are four possible actions:

| | | |
|---|---|---|
| 1) | Transition | Push the new state specified by the action, and get another token. |
| 2) | R reduction l | Pop SIZE(RHS(1))-1 states from the stack and use LHS(1) as the next input token. OUTPUT 1. |
| 3) | L reduction j | Pop SIZE (RHS(j)) states from the stack and use LHS(j) as the next input token. Return the current input token to the input stream (it was used as look-ahead). OUTPUT j. |

| | | |
|---|---|---|
| 4) | Error | Signal a syntax error and call compiler to invoke error recovery or correction. |

where "OUTPUT 1" means "reduction 1 has been recognized", and 1 is sent to the compiler.

The parsing algorithm is:

```
Push State 1 on the stack.
Get the first input token.
DO ACTION (state on top of stack, input token)
UNTIL stack is empty.
```

We have written an ALGOL 60 program which produces tables of these actions, given a grammer in BNF as input. An example of the parsing algorithm is given at the end of the paper.

## Parser Implementation

The processor is 8 bits wide (four 2-bit slices). This width was chosen since that was judged large enough to contain the information stored in the RAM. The RAM is 10 bits wide, but the 2-bit action code is not returned to the CPE. It is directly connected to a latch in the micro-sequencer, thus speeding up processing. The only quantities that are larger than 8 bits are the addresses for the RAM; they are calculated byte-serially and held by a latch until all 13 bits are present (a look-ahead carry generator is used to increase addition speeds). The CPE has two outputs, one used for sending addresses to the RAM, and the other connected to the parser output latch. The 3002 CPE also has 2 sets of inputs, one for the RAM data, and one for input from the scanner. The latter is latched and can be interrogated to determine whether data is present. Internal registers of the CPE are used to save frequently accessed quantities: the top of stack address, a pointer to the (LHS,RHS length) pairs;

a pointer to the current state, and the current token. An overall view of the parser architecture is shown in figure 4.

The instruction set of the 3002 CPE is powerful enough for this application, and contains features which made space efficient programming easy. We used a 24-bit instruction for the parsing processor. The CPE's ability to mask data on the input bus proved to be very useful. This mask was present in each instruction, and could be used to introduce constants into the CPE. The instructions also contain the jump field, function, RAM read/write enable and latch select bits. The jump field of the 3001 micro-sequencer was at first thought to be a major drawback, but the resulting code turned out to be quite dense. The code for the parser occupies only half of each 8 x 512-bit ROM.

<u>Using the Parser</u>

To use the parser, one first specifies the grammar in BNF and modifies it, if necessary, until it is in LALR(1) form (typically a rather simple process). The tables are then encoded to distinguish the actions (upper 2 bits distinguish the type of action, lower 8 bits give the reduction or state), and converted into an array of lists. The LHS and RHS lengths are determined, and the tables are loaded into the RAM. The parser is begun with a START pulse, awaits tokens from the scanner, and begins parsing. The parser outputs a stream of reduction numbers, and issues a code of 0 to signal the end of the parse. There are four error codes which are returned as pseudo-reductions with very high numbers.

| ERROR CODE | MEANING |
|---|---|
| 255 | Parse is over (e.g., an entire correct program has been recognized), but the scanner is still sending more tokens. The parser continues to output 255 until an end of input is received. |

| ERROR CODE | MEANING |
|---|---|
| 254 | Scanner error. Enter RESTART mode. |
| 253 | Syntax error. Enter RESTART mode. |
| 252 | Stack overflow in the RAM. This error is fatal, and only total restart is possible. |

The user may wish to modify the stack before restarting the parse for error recovery. For this reason the address of the top of stack is saved in the RAM (pointed to by locations 0 and 1), and is reread from there by the parsing unit on a RESTART.

The RAM organization is described in figure 2. As mentioned earlier, locations 0 and 1 contain the address of the bottom of stack, which contains the current top of stack address. Next are the pointers to the beginning of the (token,action) list for each state, followed by the (LHS,RHS length) array The (token,action) pairs follow, with each list terminated by a zero action. The top of stack address, and the stack fill the rest of the RAM. There are 20 4R-bit memory planes in the RAM, providing sufficient space for typical production compilers. For example, the PASCAL compiler under development at the University of Wisconsin (FL, 1977) uses a software LALR(1) parser. There are 222 productions, 204 states, and the matrix has 1900 non-zero entries. This means that approximately 5000 bytes of RAM would be used for tables in our design, leaving over 3K bytes for the stack. Since the stack depth typically never exceeds 30, this allows sufficient space for "practical" compilers.

<u>EVALUATION</u>

A 100 nanosecond clock was chosen for both the scanner and the parser. In the scanner, between 20 and 50 micro-instructions are needed to process each input character (assuming no delays due to the output buffers). Therefore, the scanner requires a new input character every 2 to 5 microseconds. This is a modest rate for an interleaved main memory, so it is expected that the FIFO input buffer will always be full. If we assume an average of 10 characters per token, the scanner can deliver tokens approximately every 50 microseconds. This is an effective rate of 20,000 tokens per second, or 2500 eighty-character source code lines per second.

A typical fast compiler running on a large computer will compile 10,000 source lines per minute. Since most compilers bottleneck on the scanner, breaking this bottleneck will speed up the compiler so much that other components can be coded in a less than optimal manner (e.g., in a high level language with run-time checks).

Following are the times calculated for each phase of the main parsing loop (in nanoseconds):

| | |
|---|---|
| Get a token | 600 |
| Get the list for a state | 1500 |
| Find the action | 900+1000(# of entries searched) |
| Transition to a new state | 600 |
| Reduction | 420 |

These times are exclusive of any delays in receiving tokens from the scanner, or waiting for the output latch to be emptied. It is reasonable to assume that the average list has ten elements, so about five entries will be searched to get a match. There are approximately as many transitions as reductions in a program parse, so an approximation of the average time between reductions is 600 + 1500 + 900 + 1600(5) + 800 + 4200 = 16000 nanoseconds, or 16 microseconds per reduction. With this speed, it is probable that neither the scanner nor the main processor will be able to keep up with the parser. We have also freed the table space for the parser, so the compiler can run in less space (5K in PASCAL) than if the tables were resident in main memory. This would be even more important on small computers, where the address space may be limited.

Experience with the University of Wisconsin PASCAL compiler shows that about 50% of the compilation time is spent in the software scanner and parser. Thus, we conclude that if such a parsing and scanning unit were added to a main processor, compilation could be speeded up by approximately a factor of two.

## PARSING EXAMPLE

A grammar for arithmetic expressions over + and *

```
1.  ST ::= ID := E ;
2.  E  ::= E + T
3.  E  ::= T
4.  T  ::= T * P
5.  T  ::= P
6.  P  ::= ID
7.  P  ::= ( E )
```

The table produced is:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ID | 2 | | 6R | | | 6R | 6R | 6R | | | ID |
| := | | 3 | | | | | | | | | := |
| ) | | | | | 3L | | | | 7R | 2L | ) |
| ; | | | | 1R | 3L | | | | | 2L | ; |
| + | | | | 7 | 3L | | | | 7 | 2L | + |
| ( | | | 8 | | | 8 | 8 | 8 | | | ( |
| * | | | | | 8 | | | | | 8 | * |
| P | | | 5R | | | 5R | 5R | 4R | | | P |
| T | | | 5 | | | 5 | 10 | | | | T |
| E | | | 4 | | | 9 | | | | | E |

A parse of  ID := ID + ID * ID  would be:

```
             STACK  TOKEN   INPUT STREAM     OUTPUT

                 1    ID    := ID + ID * ID ;
               1 2    :=    ID + ID * ID ;
             1 2 3    ID    + ID * ID ;      6 (P <- ID)
             1 2 3    P     + ID * ID ;      5 (T <- P)
           1 2 3 5    +     ID * ID ;        3 (E <- T)
             1 2 3    E     + ID * ID ;
           1 2 3 4    +     ID * ID ;
         1 2 3 4 7    ID    * ID ;           6 (P <- T)
         1 2 3 4 7    P     * ID ;           5 (T <- P)
         1 2 3 4 7    T     * ID ;
      1 2 3 4 7 10    *     ID ;
    1 2 3 4 7 10 8    ID    ;                6 (P <- ID)
    1 2 3 4 7 10 8    P     ;                4 (T <- T * P)
         1 2 3 4 7    T     ;
      1 2 3 4 7 10    ;                      2 (E <- E + T)
             1 2 3    E     ;
           1 2 3 4    ;                      1 (ST <- ID := E ;)
             empty                           0 (end of parse)
```

The parse tree is

```
    ST
  /   \
ID  :=  E  ;
       / \
     E  +  T
     |    / \
     T   T * P
     |   |   |
     P   P   ID
     |   |
     ID  ID
```

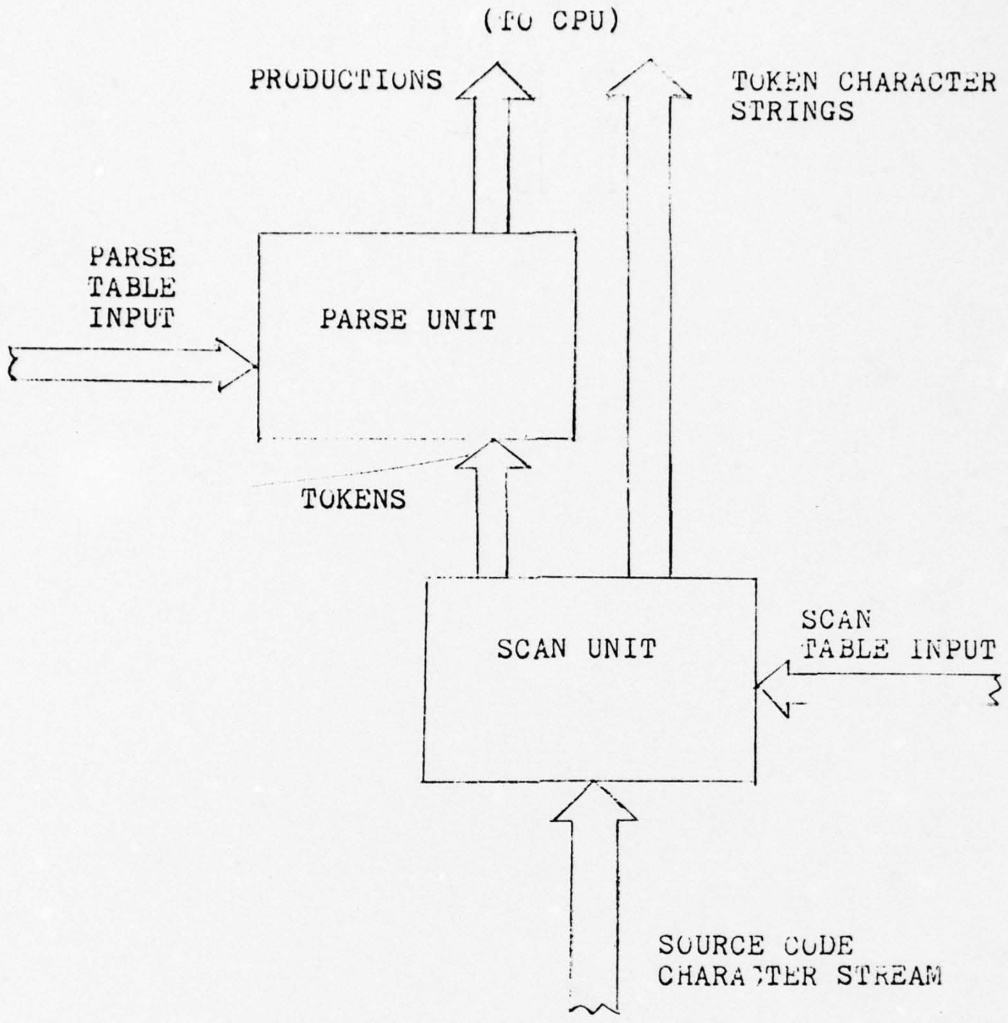
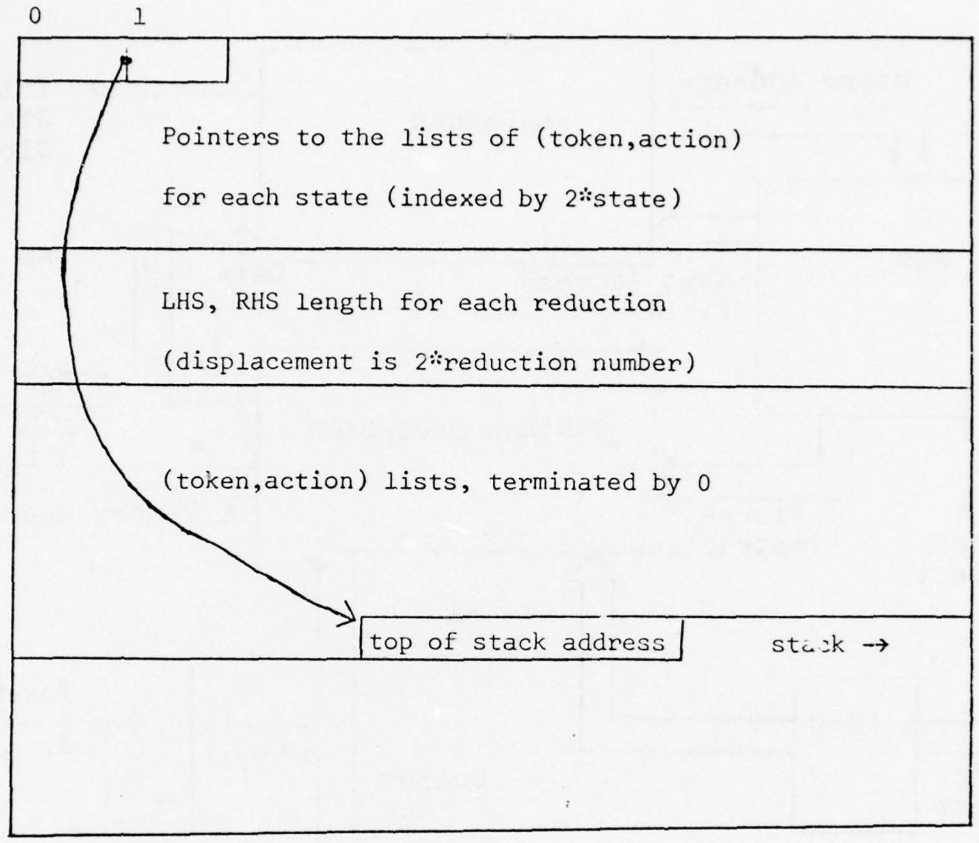


Figure 1. Pre-processor Diagram


0
1
Pointers to the lists of (token,action) for each state (indexed by 2*state)
LHS, RHS length for each reduction (displacement is 2*reduction number)
(token,action) lists, terminated by 0
top of stack address
stack →

Parser RAM organization

figure 2

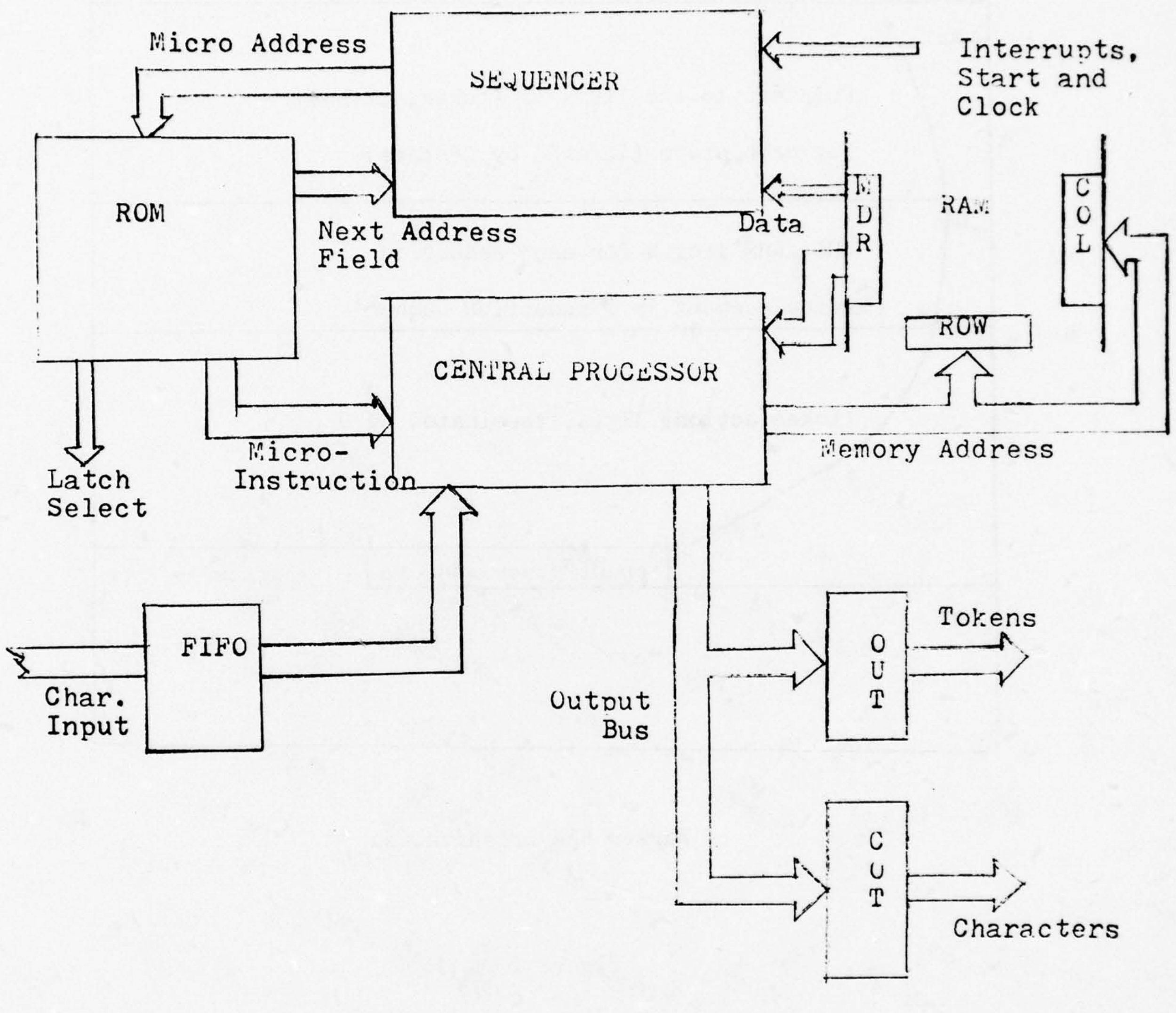
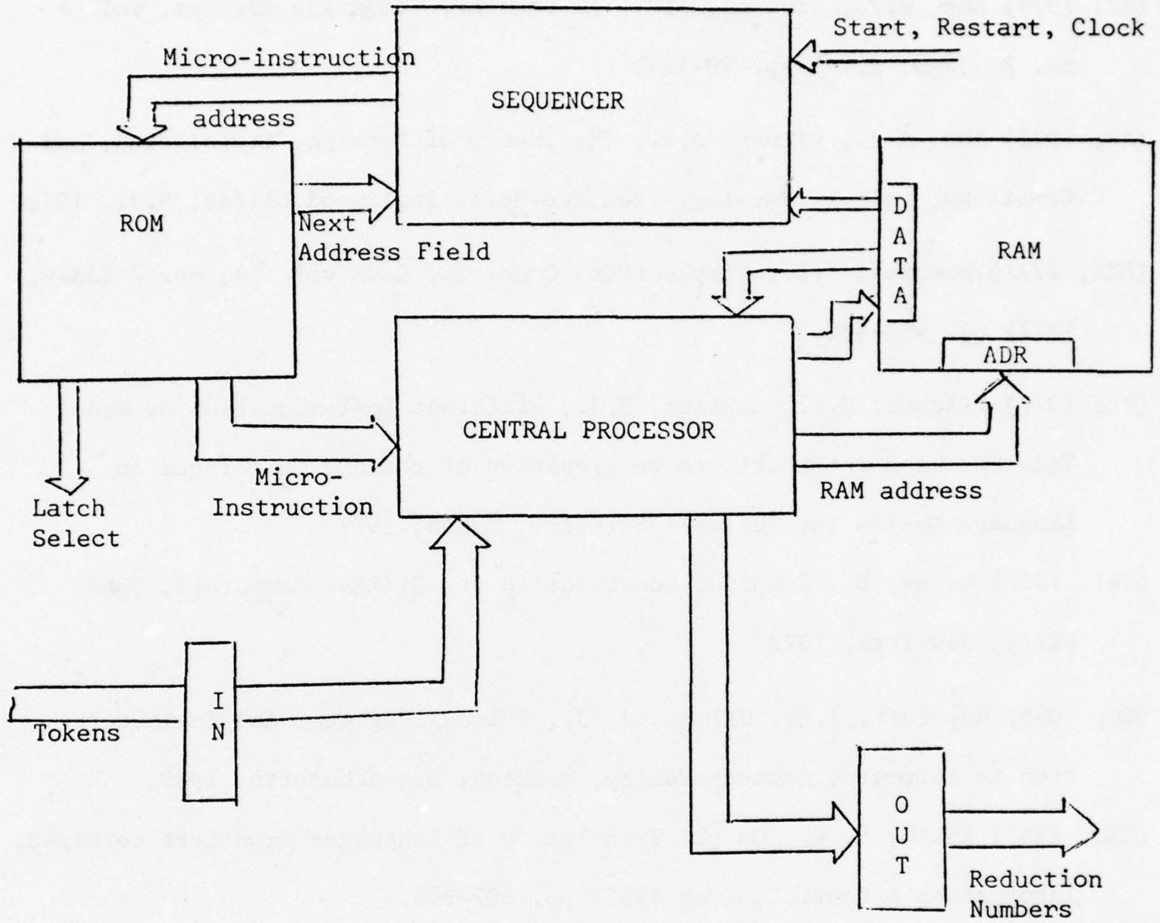


Figure 3. Gross Scan Unit Architecture



GROSS PARSER ARCHITECTURE

Figure 4

## REFERENCES

(AJ, 1974) Aho, A.V., Johnson, S.C., LR Parsing, Computing Surveys, vol. 6 no. 2 (June, 1974) pp. 99-124.

(AU, 1972) Aho, A.V., Ullman, J.D., The Theory of Parsing, Translation, and Compiling, vol. 1: Parsing, Prentice-Hall, Inglewood Cliffs, N.J., 1972

(DER, 1971) Deremer, F.L., Simple LR(k) Grammars, CACM vol. 14, no. 7 (July, 1971) pp. 453-460.

(FL, 1977) Fischer, C.N., LeBlanc, R.J., Efficient Implementation of Run-Time Checking in PASCAL, to be presented at the ACM Conference on Language Design for Reliable Software, March, 1977.

(GRI, 1972) Gries, D., Compiler Construction for Digital Computers, John Wiley, New York, 1972

(HU, 1969) Hopcroft, J.E., Ullman, J. J., Formal Languages and Their Relation to Automato, Addison-Wesley, Reading, Massachusetts, 1969.

(KNU, 1965) Knuth, D. E., On the Translation of Languages from Left to Right, Information & Control, (Oct 1965) pp. 607-639.

# A NEW LOOP STRUCTURE FOR DISTRIBUTED MICROCOMPUTING SYSTEMS

H. Jafari
Electrical Engineering Department

T. Lewis
Computer Science Department
Oregon State University
Corvallis, Oregon 97331

## ABSTRACT

This paper presents a new distributed computer network structure appropriate for a network of microprocessors. The new network structure combines advantages of a ring structure; simplicity, high line utilization, concurrent service, distributed control information, minimum delay for minimum cost, and high reliability. This is accomplished using two loops. The "inner" loop is for data transfer. It is partitioned into N buses interconnecting N microprocessors. The "outer" loop is for control information to pass along under the guidance of a bus controller. Results for simulations of contemporary proposals (Pierce, Newhall, and Reames et al.) and the new network proposed in this paper show that the new structure substantially improves throughput when compared to the other structures.

## INTRODUCTION:

Researchers have proposed distribution of low-cost computing processors throughout a network as an alternative to expensive and highly centralized computer systems (SPAN 76). The results have shown that completely distributed systems lead to a great deal of inefficiency due to increased hardware and software overhead and often fail to deliver acceptable throughput as expected. In addition a computer network introduces other complexities concerning deadlocks, network reliability, traffic regulation, and scheduling.

This paper introduces a new network topology with high throughput rate for distributed computer systems. The network has an improved response time, greater throughput, and is more reliable than the Pierce, Newhall, or Reames - Liu loop network topologies.

## I. DESIGN PHILOSOPHY

A distributed computer system interconnects several heterogeneous or homogeneous nodes which communicate with each other through network media. A heterogeneous network is a collection of architecturally different nodes while a homogeneous network is a collection of architecturally similar processor nodes.

Farber (FARB 72) lists the motivations to develop a distributed computer system as any or all of the following:

1) Modular Growth
2) System Reliability
3) Incremental Upgrading of Processor Nodes
4) Dynamic Restructing
5) Decreased Design Time
6) Ease of System Validation

In addition we include:

7) Tailored Design to the Users Needs
8) Better Throughput (Speed)
9) Less Cost

With these motivations in mind, several people have proposed and implemented a variety of network topologies in hopes of efficiently managing distributed computer systems.

The topology of the interconnections in a network is of great concern since it has a major effect on the performance of the distributed system. The most highly connected network is to connect every computer to every other directly. This involves N(N-1)/2 interconnections for N nodes and is very costly unless N is very small. A less costly topology requiring N interconnections and an additional central control processor is the star configuration. The central control computer provides node-to-node interconnection by switching from one node interconnection pattern to another upon demand. Furthermore, each distributed star computer system can be connected to another star computer system by connecting the two central control computers together, and with appropriate control algorithms, this will allow any node in either subnetwork to communicate with any others.

A problem with star network computer distribution is reliability of the system, for as soon as the central computer exhibits faulty functions, the whole system breaks down. In addition, the central control processor is an overhead cost added to the whole system. If the number of nodes around the central processor is small, then the advantage of this system is its speed, also because the links between computers are bidirectional, the system has a very good throughput. We will not include the star network in the work reported here because of its poor reliability (STRE 76).

Another philosophy is to connect all the processor nodes in a loop or ring configuration. This is called a loosely coupled connection since each node is connected to others by only two links, an input link which comes to the node and an output link that goes away from the node. Loop systems are attractive for mini-micro computer networks due to their possible high line utilization and because they are simple. This last philosophy has attracted the attention of many researchers who have designed a variety of network systems based on the simplicity of a loop. The new loop structure will allow more parallel communications between nodes, while taking advantage of loop simplicity.

## II. PREVIOUS LOOP CONTROL ALGORITHM

The first loop structure system was suggested by NEWHALL (FARM 69). In the NEWHALL loop a round-robin control passing mechanism circulates around the loop and allows only one node at a time to transmit one or more messages through the loop. Therefore, the rest of the nodes have to wait and this causes a queuing time in sending the messages which limits the achievable loop utilization.

A version of a loop discipline similar to the NEWHALL discipline is allowed with IBM's SDLC (DONN 74) (or with the largely equivalent HDLC (DAVI 73)). In this discipline a central controller originally sends a poll command around the loop. The first attached device wishing to transmit is thereby enabled to transmit. This device then ends its transmission by passing the poll on, so that control passes around the loop in a manner similar to the behavior of a NEWHALL loop. This variation is not explicitly studied here because of its similarity to the NEWHALL loop. On the other hand, Pierce (PIER 72) introduced a new mechanism that improves network utilization by time multiplexing the loop. That is, the information sent around the loop is divided into fixed-size packets and to send a message, each node checks for an empty packet before transferring all or part of its message. If a message is smaller than the fixed-size packet, the excess space is wasted. If the message is too large to fit the packet, then the message is broken into two or more packet-sized messages. When a processor node transmits a message, it must first check whether the next packet or time slot passing by it is empty. If it is, control will pass to the processor nodes transmitter to see if there is any information to be transmitted. In case the packet is not empty, the processor node checks to see if the destination address in the packet matches the node address. If so, the processor node transfers the packet information into its buffer. If the packet address does not match the processor node address, then the processor simply passes this packet to the next node. The transmission mechanism is as simple as waiting for the beginning of an empty slot and filling it with a packet, but disadvantages of this system include:

a) problem of dividing messages into packets
b) problem of packet reassembly which occurs when messages are divided into packets and then sent separately, so a sorting problem arises.
c) messages do not always fit into a fixed number of packets, so there are some partially empty packets with corresponding waste of network capacity.

Therefore, neither Newhall or Pierce loops make very efficient use of loop topology. Reames and Liu (REAM 75) introduced a new message transmission mechanism called DLCN (Distributed Loop Computer Network) which allows multiple messages in the loop as the Pierce loop does and messages of variable length as the Newhall loop permits.

DLCN incorporates a variable length shift register before each node's transmitter, see Figure 1. A message can be transmitted through the loop whenever no other message transmission is already in progress, or no other messages have started passing that node. In this case, the variable shift register provides a delay in the incoming message equal in size to at least the size of the message to be inserted. Once an incoming message has been delayed in this manner, it is transmitted ahead of any incoming messages which are in turn delayed during the time needed to transmit. The contents of the variable length shift register will gradually decrease in length and finally be eliminated if there is not enough traffic. DLCN actually combines Newhall and Pierce loop advantages by allowing simultaneous message arrival with message transmission, and also provides automatic traffic regulation based on observed system load, but DLCN favors infrequent requests while delaying more frequent requests for network service.

A disadvantage of the DLCN is the complexity of interface mechanism and, therefore, the cost to build such an interface. Secondly, inserting a variable shift register at each node lowers the reliability of the overall loop since it adds one new possible failure mode. Also, when the number of nodes in the loop increases, eventually the queuing time will increase drastically. This limits the number of nodes inserted in a loop.

Potvin (POTV 71) introduced a generalized distributed computer system called the star ring system. It combines the control feature of a loop network with the message transmission features of a star network. It is somewhat similar to Newhall's technique for passing control along its loop and in its method of time multiplexing message transmission. The system is restricted by the number of nodes on the loop because the central star ring is common to all the nodes and, therefore, not more than two nodes can talk to each other at any time. This slows the throughput of the system by a great amount. Potvin considers only a very small number of nodes in the network.

All the above communication loops suffer from the following common shortcomings in addition to the problems discussed above.

1) The stream of data is in one direction and therefore, sometimes the transmission of data from one node to its neighbor node takes place through the rest of the nodes causing more delay and less reliability than necessary.
2) If a node starts sending a stream of messages to another node it will block out all other transmissions and network performance will decrease by a great amount. Thus, the networks mentioned above are sensitive to local demands that affect the performance of all nodes.
3) If there are errors in the address fields of the message and/or a node fails to function properly, messages will saturate the loop, in all the above systems. Several different techniques have been used to recover from errors, but this eventually slows down loop communication.

4) If there is a failure in the loop, the whole network will fail to operate.

A new experimental loop is proposed that will enable the whole network to recover from the above shortcomings. The concept is to distribute data and control into two different loops (a data loop and a controller loop). The data loop is actually a segmented loop consisting of a single segment connecting nodes. Each node is interfaced to the loops by a switch that may be turned "on" or "off". The control loop operates by a simple arbiter, which accepts requests for communication, decides the minimum route, and sets up the data paths between nodes by turning appropriate switches "on" and non-appropriate switches "off".

## III. DESCRIPTION OF NEW LOOP NETWORK

We suggest a modified loop network in which control messages and data messages are transferred through two different communication lines. This adds flexibility to the network for very little increase in cost. The loop network system is configured from four different components:

1) control line loop
2) data line loop
3) processor nodes
4) a special processor node dedicated to line control.

The control line loop employs a polling technique to start and stop the transfer of messages from a source node to a destination node. Transmission is accomplished through a "double handshake" where a request to send is followed by an acknowledgement that the message has been received. In particular, there are two different possible types of messages, SYN/ACK and Relay Control which can be sent over the control line.

SYN/ACK: When a node desires to communicate with another node (SYN), or respond to end of communication (ACK), then it will send a message to the controller containing the address of the source node and the address of the destination node along with the command (either SYN or ACK) to be performed by the controller. Messages of this type have the format shown in Figure 2(A).

Relay Control: Messages sent from the controller to a source or destination to inform the node that a message is being sent to it (destination), or that a message has been received by the destination node (source), or directing other nodes to position their data switches to bypass the data and allow it to continue along the data loop until reaching its intended destination. The messages of this type are shown in Figure 2(B).

The data line loop transfers all the data messages from any source node to any destination node through a minimum route which has already been set up by the controller as explained above. The data line loop illustrated in Figure (3) is interfaced to each node through a three-way switch at each node which enables the node to connect segments of the data line together and either bypass the node or connect the node to the data loop so that the node can receive or send data. The controller sets the three-way switches before each data transmission is allowed. For example, if Node 1 of Figure (4A) is to send data messages to Node 3, then the switches and data segments are connected in one of the configurations shown in Figure (4). Observe that the connection of segments of the data loop permit partial use of the entire data loop network Figure (4). Remaining segments of the data loop are available for concurrent data transmission to other nodes in the system. Therefore, simultaneous transfer over non-interfering segments of the network is quite possible. The combined effect of redundant alternate paths and concurrent transmission over non-interfering segments of the loop adds to the network reliability and throughput.

The partitionable loop structure described above is a general structure. In addition to the loop topology studied here, there is also the potential for other configurations. The topology of a specific network may require high-speed transmission between two or more nodes, depending upon the needs of these two processor nodes. In such a special case, it may be expedient to include additional "express" buses to supplement the basic loop. This can be done, for example, as shown in Figure (5), by merely increasing the capability of control line switches at these nodes. In the examples of Figure (5), supplemental data buses may be used to establish high bandwidth communication between Node 1 and Node 4. Alternatively, the response time of communication between Nodes 4 and 2 may justify an additional data line as shown in Figure (5B).

Processor nodes are configured from four elements:

A. A node control mechanism to perform data loop and control loop functions.

B. Control switches to switch the data lines.

C. Transmitter and receiver.

D. Terminal processor which may be a simple I/O device, a microcomputer, or an interface to another network.

Figure (6) illustrates these four elements. Each node control mechanism provides timing control, message detection, decoding and encoding of messages, controlling the data switches, transmitter and receiver control, and communication with its node terminal.

The control switch is a modular unit easily extendable through hardware changes; for instance, a control switch can control two data segments along with the receiver and transmitter. If the number of data segments interfaced to the node increases, the complexity of the switches will increase in a modular manner.

A simple transmitter-receiver can be time multiplexed or separated from each other by using separate channels which adds to complexity to the control switches. Figure (7) shows both a simple and more complex transmitter receiver section.

The loop network interface is designed as an "intelligent interface" so that no assumption about the processor terminal is needed. Any device may be plugged into the loop network regardless of its sophistication. All the control needed for any terminal to talk to the receiver-transmitter section is provided by the node control, thus allowing terminals to be of any type. The intelligence of the node controller is easily provided by a low-cost microprocessor and PROM.

The loop controller functions are as follows:

A. Sends and receives control messages to and from control line.

B. Schedules node communications.

C. Finds the minimum path between the nodes which are to communicate.

D. Provides a timing mechanism.

The control messages have the formats of Figure 2(A) or Figure 2(B). The controller decodes or encodes them by managing the right timing. Scheduling of nodal communication may be by any scheduling algorithm as LIFO, FIFO, round robin, or shortest-messages-first. For the routing algorithm, any method can be considered, but since all the needed information is within the controller, routing can be tailored to special applications of the network. The timing mechanism can be part of the controller's function to synchronize all the nodes or it can be varied in each individual node. Therefore, nodes can work synchronously or asynchronously. The function of the controller is flowcharted in Figure (8). The functions of the network controller are very straightforward and can be performed by any node in the network. We will assume a special control node microprocessor is used to perform the control functions for the entire network. In the comparisons to follow, we will include this special-purpose control node as an overhead cost, but it should be pointed out that the control functions required by the proposed loop can be carried out by any node. In terms of reliability, this means that failure of the control node does not imply failure of the entire network, because control can be passed to another (working) node on the loop.

## IV. SIMULATION RESULTS

We modeled our simulation study after the work of Reames and Liu (REAM 75). They simulated the DLCN (Distributed Loop Computer Network), Newhall Loop, and Pierce Network. The results obtained in our study will be compared with their results. Our results will extend their results to provide an evaluation of all four network topologies. In the DLCN simulation model, the length of the shift register interface to the loop was 512 characters. For the Pierce model, Reames and Liu selected a packet size of 36 characters. This is an optimal packet size obtained by minimizing the product of average number of packets times the packet size. In the Newhall network, they simulated passing the control token only when the queue of messages in that node is empty instead of passing one message at a time at each node. This produces a shorter total message transmit time for the Newhall network.

For all of the systems simulated by Liu and Reames, message length has a truncated negative exponential distribution with a mean of 50 characters, minimum of 10, and maximum of 512 characters of which the first nine characters are control characters. Message arrival time obeys the Poisson distribution, and the number of nodes is 6.

For the new experimental loop, the message length and message arrival statistics, and the number of nodes are the same as above. There is no need for control messages along with data in this new system. For reliability purposes, we used the same number of characters by including control characters with the data. The messages in this system can be of any length without hardware or software constraints.

The scheduling algorithm is simple FIFO and the routing algorithm is to simply find the minimum path between two nodes in either direction. If two paths have the same length, the clockwise direction is arbitrarily chosen. The new loop network improves throughput when employing these simple algorithms for scheduling and routing.

Table 1 shows the average interarrival rate, data line usage, waiting time for each message to be transmitted, transmission time total transmission time, and control line usage for the new experimental network, as well as for the other three networks. Figure (9) shows the variation of mean total message transmission time versus mean arrival rate for all four networks. Figure (10) shows the changes in line utilization versus changes in interarrival rate for all systems, which indicates the load of the system, and finally Figure (11) is a graph of mean control line utilization versus the mean interarrival rate for new experimental systems, only.

From Table 1, we see the Pierce and Newhall loops and new experimental loop have almost a constant transmission time for any load on the system (46 time units per packet for Pierce loop, and 63 time units per message for Newhall loop, and 52 time units for the new experimental loop). This is due to a constant delay in the transmission lines for Pierce and Newhall systems. For the new experimental loop, there is no delay in transmission line. Transmission time is equal to the transfer time of the characters in a message. For DLCN, message transmission time is variable and as soon as the arrival rate increases ( that is, the system load increases) then the shift register delay line time will increase leading to an increase in transmission time proportional to system load. On the other hand, the queuing time at each node will not increase as fast as transmission time since whenever a message is ready to go in the loop, the node will insert the variable delay shift register in the loop and then the message does not have to wait longer. This explains why DLCN is faster than the Pierce and Newhall loops. The superior performance of the new experimental loop is due to multiple concurrent transmission, variable message length without any additional hardware or software overhead, and the ability to select the shortest path from the bidirectional segments of the loop. As we see from Figure (9), total transmission time for Newhall, DLCN, and the new experimental loop is the same for very low system load. But as soon as the load on the system goes higher, the total transmission time for the new experimental loop shows improvement over the others. In the Pierce loop, a message always has a mean wait equal to one-half of the packet size and must then be transmitted in several packets. For this reason, the Pierce loop can not compete with the others for low systems loads. As soon as the system load goes higher, the Pierce loop exhibits concurrency (simultaneous packets on the loop) and its performance improves over the Newhall loop which shows its inherent serial nature leading to poorer performance.

In our new experimental loop there is a minimum queuing time for SYN/ACK and relay control messages. For low loading of the network this overhead shows up as a significant part of the overall delay, but since these two control messages cause a constant average delay they contribute a smaller proportion of the delay as the network load increases. Typically messages are queued before being transmitted and the delay due to control messages is overlapped with the fixed control message's queuing time.

The greatest advantage of the new network is that segments of the loop can be activated simultaneously. The added concurrency of the new loop explains its increased throughput when compared with the other networks. From Figure (10) we see the mean line utilization is very low for all the networks. As system load increases the line utilization for Newhall network levels off at about 50 percent. For Pierce and DLCN systems, line utilization increases as system load increases. However, when the loop is utilized up to its maximum, the waiting time will increase drastically. The proposed network requires nearly half of the line utilization of the other loops simulated. Figure (11) shows a linear relationship between the mean control line usage and system load. This is due to constant delay for SYN/ACK messages, however the relay control message is of variable length, (changes are within 7 percent).

## CONCLUSION:

The main goal of this work was to improve the throughput of a microcomputer network using a flexible, simple, and reliable loop topology.

The results of our simulation have shown that completely decentralizing microcomputers leads to a decrease in throughput compared to the expected throughput of n processors. The loss in throughput resulting from networking multiple processors can be partially compensated for by careful design of the network and its interfaces. Reliability can be achieved by permitting any node to take over the controller's job.

The hardware implementations given for the interface and line controller show compatibility of this system with microprocessor technology. Because microprocessors are low cost, this type of network can be constructed inexpensively.

Future research in this area will be done using different scheduling algorithms for the controller, using a different number of nodes, and with different types of loop structures. Also an investigation of a mathematical model for such a loop structure, as has been done in the past for other loop structures is needed. (SPRA 72), (HAYE 74), (KONH 72), and (KAYE 72).

ACKNOWLEDGEMENT:

The authors wish to express their appreciation to J. Spragins for his advice and consultation during this research, and to C. Reames for supplying us with his GPSS simulation program. Also we are thankful for the participation of W. R. Adrion.

REFERENCES:

DAVI 73 - Davies, D. W., Barber, D. L. A., "Communication Networks for Computers", John Wiley, London, 1973, pp. 234,235.

DONN 74 - Donnan, R. A., Kersey, J. R., "Synchronous Data Link Control: A Perspective", IBM Systems Journal, 13, No. 2, 1974, pp. 140-162.

FARB 72 - Farber, D. J., Larson, K., "The Structure of a Distributed Computer System - The Communication System", Proc. Symp. on Computer Communications, Networks and Teletraffic, Polytechnic Institute of Brooklyn Press, 1972, pp. 21-27.

FARM 69 - Farmer, W. W., Newhall, E. E., "An Experimental Distributed Switching System to Handle Bursty Computer Traffic", Proc. ACM Symposium. "Problems in the Optimization of Data Communications System", Pine Mtn. Georgia, Oct. 1969.

HAYE 74 - Hayes, J. F., "Performance Models of an Experimental Computer Communication Network", BSTJ, Vol. 53, No. 2, 1974, pp. 225-259.

KAYE 72 - Kaye, A. R., "Analysis of a Distributed Control Loop for Data Transmission", Proc. Symp. on Computer Communication Networks and Teletraffic, Polytechnic Institute of Brooklyn Press, 1972, pp. 47-58.

KONH 72 - Konheim, A. L., Meister, B., "Service in a Loop System", Journal ACM, Vol. 19, No. 1, 1972, pp. 92-108.

PIER 72 - Pierce, J. R., "Network for Block Switching of Data", BSTJ, Vol. 51, No. 6, 1972, pp. 1133-1145.

POTV 71 - Potvin, J. N. T., "The Star-Ring System of Loosely Coupled Digital Devices", University of Toronto, Computer Systems Research Group Report No. 7, 1971.

REAM 75 - Reames, C. C., Liu, M. T., "Design and Simulation of the Distributed Loop Computer Network (DLCN)", in Proc. 3rd Annual Symposium on Computer Architecture, Clearwater, Florida, January 1975, pp. 7-12.

SPAN 76 - Spang, III, H. A., "Distributed Computer Systems for Control", General Electric Technical Information Series Report No. 76CRD049, April 1976.

SPRA 72 - Spragins, J. D., "Loop Transmission Systems - Mean Value Analysis", I.E.E.E. Trans. Communications, Vol. COM-20, No. 3, 1972, pp. 592-602.

STRE 76 - Strevens, C. W., "Current Research in Computer Network", ACM Computer Communication Review, April 1976, Vol. 6, No. 2, pp. 13-40.

+ NEW LOOP
x DLCN
- PIERCE
* NEWHALL

T A B L E 1

| INTERARRIVAL RATE | DATA LINE USAGE | QUEUING TIME | TRANSMISSION TIME | TOTAL TRANSMISSION TIME | CONTROL LINE USAGE | |
|---|---|---|---|---|---|---|
| 3600. | .025 | 19.42 | 51.36 | 70.78 | 0.020 | + |
| 1500. | .065 | 30.33 | 53.58 | 83.41 | 0.051 | + |
| 900. | .100 | 39.66 | 51.28 | 90.94 | 0.083 | + |
| 600. | .155 | 61.00 | 52.36 | 113.36 | 0.123 | + |
| 480. | .185 | 773.25 | 50.25 | 123.50 | 0.154 | + |
| 420. | .218 | 107.79 | 51.86 | 159.80 | 0.179 | + |
| 340. | .275 | 160.00 | 51.27 | 211.27 | 0.222 | + |
| 300. | .324 | 266.78 | 54.28 | 321.06 | 0.247 | + |
| 270. | .340 | 335.58 | 51.09 | 386.67 | 0.276 | + |
| 240. | .387 | 596.43 | 51.51 | 648.95 | 0.307 | + |
| 220. | .432 | 1134.11 | 52.77 | 1186.88 | 0.342 | + |
| 3600. | .056 | 2.10 | 58.60 | 74.50 | [illegible] | x |
| 1500. | .138 | 6.40 | 61.30 | 86.70 | [illegible] | x |
| 900. | .235 | 12.20 | 67.80 | 103.90 | [illegible] | x |
| 600. | .365 | 19.40 | 79.60 | 136.10 | 0.--- | x |
| 480. | .474 | 30.10 | 102.10 | 175.20 | --- | x |
| 420. | .543 | 39.90 | 115.90 | 210.20 | --- | x |
| 342. | .677 | 64.20 | 150.80 | 297.70 | --- | x |
| 300. | .759 | 101.60 | 210.30 | 404.00 | --- | x |
| 270. | .844 | 181.50 | 332.70 | 648.40 | --- | x |
| 240. | .937 | 303.10 | 648.90 | 900.60 | --- | x |
| 2700. | .098 | 10.90 | 104.30 | 115.20 | --- | - |
| 1800. | .147 | 18.70 | 105.70 | 124.40 | --- | - |
| 1200. | .200 | 27.90 | 105.80 | 133.70 | --- | - |
| 900. | .293 | 47.10 | 105.00 | 152.10 | --- | - |
| 720. | .367 | 69.10 | 105.00 | 174.10 | --- | - |
| 600. | .430 | 74.90 | 106.00 | 180.90 | --- | - |
| 540. | .479 | 119.10 | 106.20 | 215.30 | --- | - |
| 480. | .513 | 1148.40 | 103.40 | 251.80 | --- | - |
| 420. | .633 | 215.60 | 110.50 | 326.10 | --- | - |
| 360. | .717 | 257.70 | 107.60 | 365.30 | --- | - |
| 330. | .762 | 360.90 | 102.80 | 463.70 | --- | - |
| 300. | .801 | 587.20 | 103.50 | 690.70 | --- | - |
| 270. | .935 | 1412.00 | 99.00 | 1511.00 | --- | - |
| 2100. | .153 | 15.30 | 62.60 | 77.80 | --- | * |
| 1500. | .183 | 21.10 | 73.30 | 84.40 | --- | * |
| 900. | .242 | 38.60 | 62.40 | 101.00 | --- | * |
| 600. | .328 | 75.50 | 62.20 | 137.70 | --- | * |
| 480. | .378 | 135.20 | 63.30 | 198.50 | --- | * |
| 420. | .424 | 283.60 | 62.90 | 346.50 | --- | * |
| 360. | .487 | 611.60 | 63.80 | 675.40 | --- | * |
| 330. | .518 | 3210.00 | 59.00 | 3269.00 | --- | * |
| 300. | .511 | 6564.00 | 68.00 | 6632.00 | --- | * |

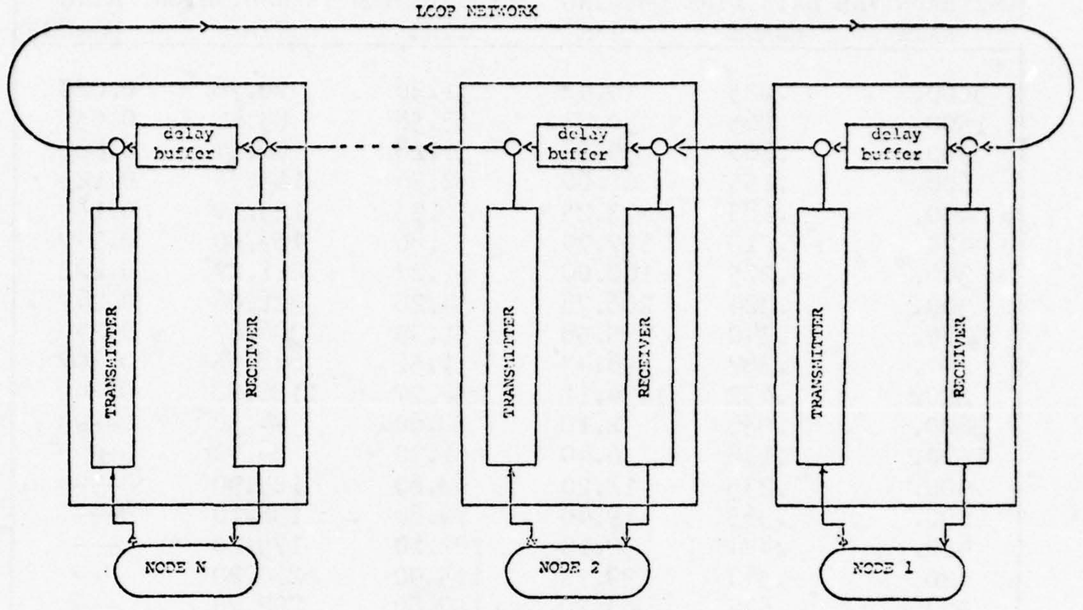


FIGURE 1

| polling information | source node | dest. node | command |
|---|---|---|---|

FIG. (2a)

| polling information | node address | comm-and | ≀ | node address | comm-and |
|---|---|---|---|---|---|

FIG. (2b)

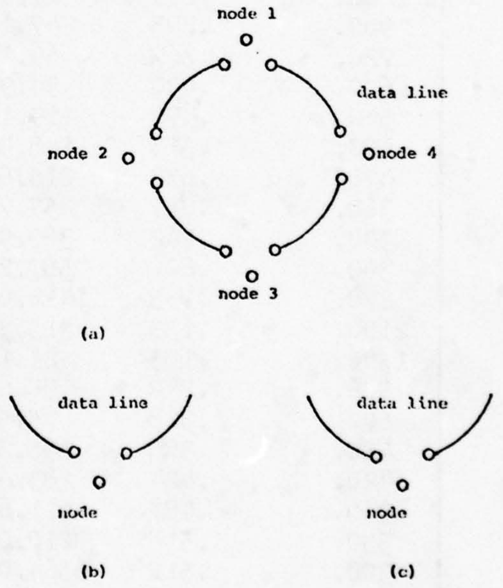


FIGURE 3

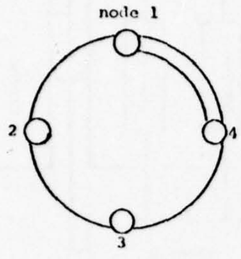


FIG. (5a)

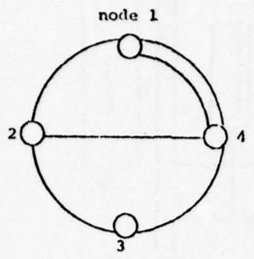


FIG. (5b)

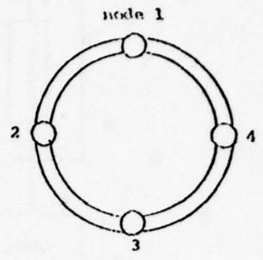


FIG. (5c)

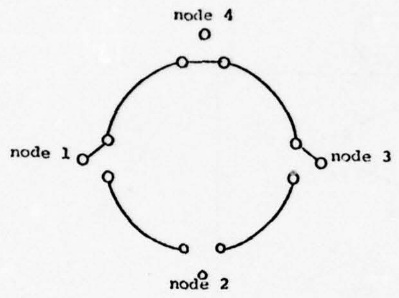


FIG. (4a)

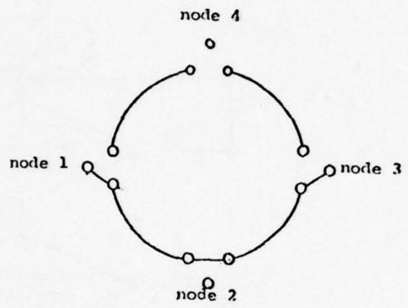
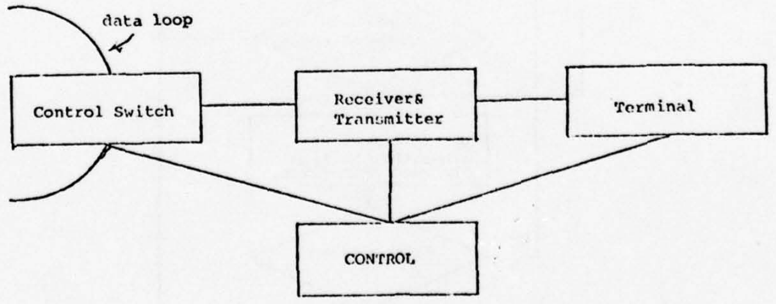
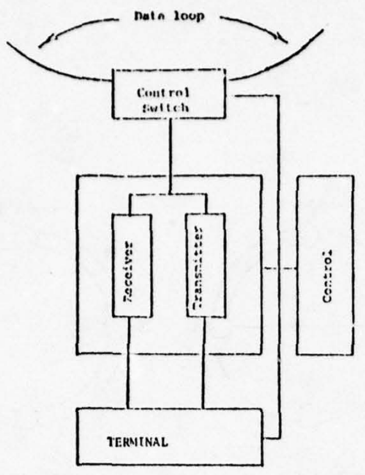


FIG. (4b)





FIG_(7a)

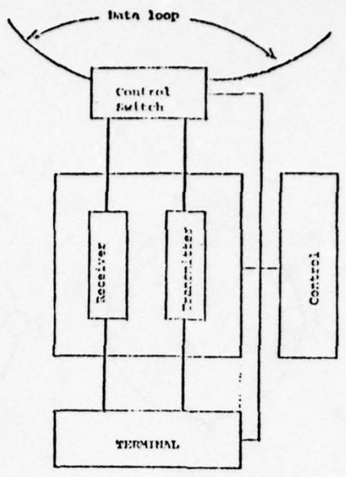


FIG. (7b)

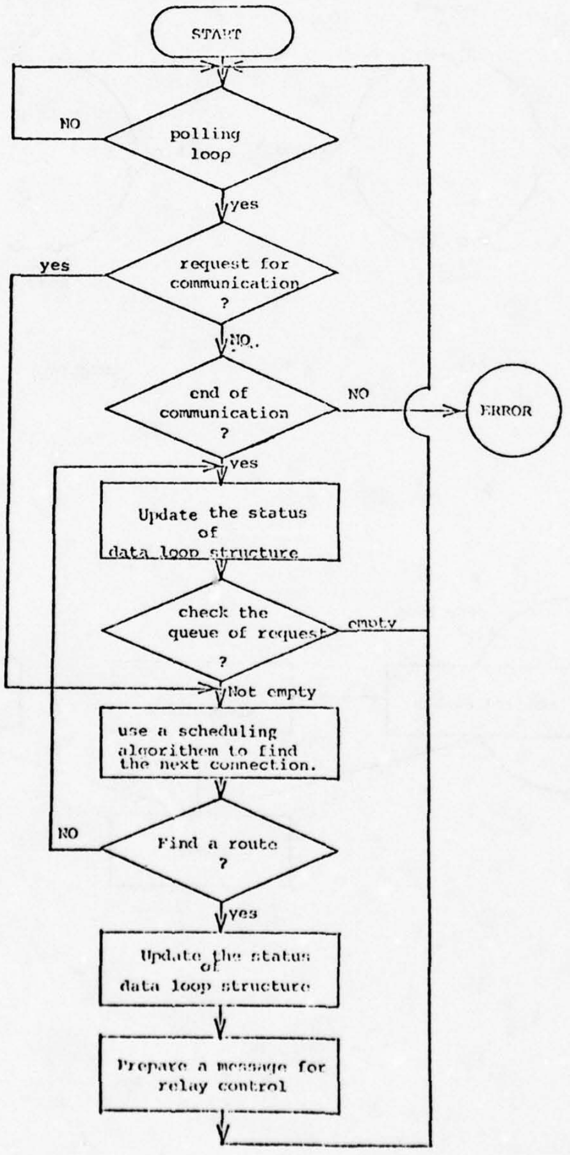

START
polling loop
NO
yes
request for communication ?
yes
NO
end of communication ?
NO
ERROR
yes
Update the status of data loop structure
check the queue of request ?
empty
Not empty
use a scheduling algorithem to find the next connection.
Find a route ?
NO
yes
Update the status of data loop structure
Prepare a message for relay control

FIGURE 8

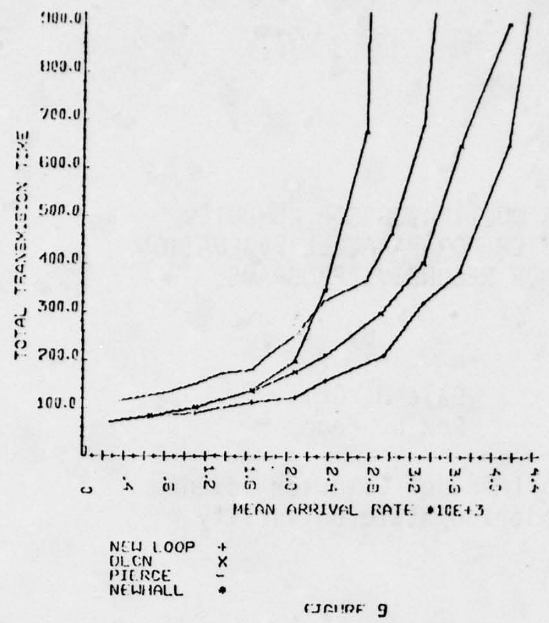


FIGURE 9

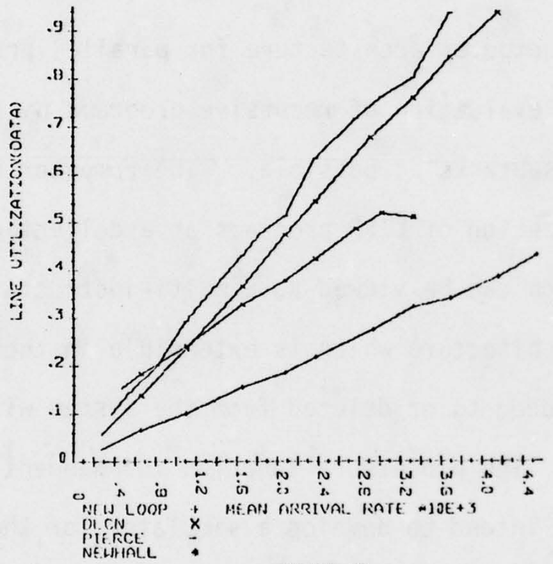


FIGURE 10

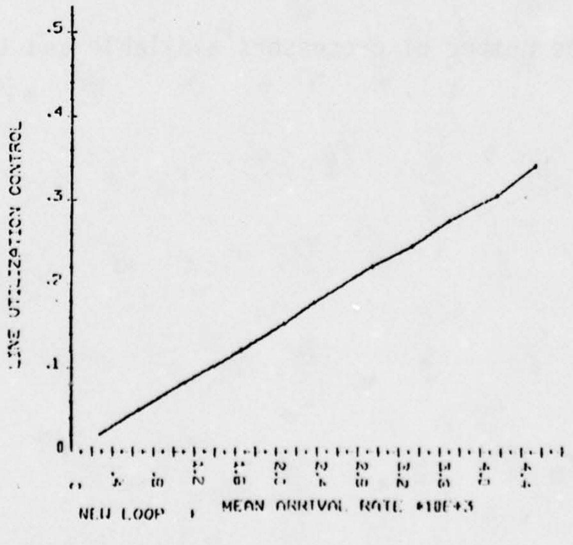


FIGURE 11

# A MULTIPROCESSOR COMPUTER SYSTEM FOR PARALLEL EVALUATION OF RECURSIVE PROGRAMS

by

Dale H. Grit
Rex L. Page

Department of Computer Science
Colorado State University

## Abstract

We describe a computer architecture for parallel processing centered around the parallel evaluation of recursive programs by simultaneously performing as many subtasks as possible. The computer system is based on the parallel execution of LISP programs as a collection of asychronous subtasks. The design can be viewed as a multi-instruction, multi-data (MIMD) data-flow architecture which is extensible in the sense that processors can be added to or deleted from the system without disturbing the overall design. The processors function independently using a shared central memory. We intend to develop a simulator for the system in order to collect statistics on the speed-up attainable with this architecture as a function of the number of processors available and the kind of tasks being performed.

## 1. Introduction

1.1 General. LSI technology has provided us with large numbers of inexpensive microprocessors. In order to develop the full potential of this technology, we must move beyond the concepts of sequential control for programs and centralized control for computer systems. Because we are reaching the technological limits of processor speeds, parallelism and distributed processing appear to be the techniques by which the speed of computation will be increased. Extensive use of microprocessors in the design of computer systems makes it possible to rank CPU components among the relatively cheap parts of the system. This calls for a new viewpoint in design and makes the consideration of radically new and different architectures worthwhile. The computer system we will describe uses a large number of microprocessors working in an asynchronous mode in order to attain a significant degree of parellelism in the computations being performed. Furthermore, the system is able to take advantage of parallel computation without requiring programmers to use new and specialized techniques in designing algorithms for the system.

This latter point is significant because the cost of software has continued to increase while reliability of software has become a very serious problem. If this is true for sequential programs, which we have been coding since the advent of computing, the problems of reliability and production costs will be even more prevalent in programming parallel systems where the programmer must tailor his algorithms to a specific parallel machine organization in order to take full advantage of the system. Currently, there are very few languages available which support parallel programming. Parallel notation is still at a primitive level. A notation for parallelism in algorithms which is independent of particular machines is desirable. A notation which doesn't require the programmer to explicitly determine the parallelism is even more desirable. Then the programmer can turn his attention to the proper design of the program and leave it to the computer system to discover parallelism in the algorithm. Since purely recursive programs make this possible, we want to investigate the degree of speed-up attainable through parallel evaluation of recursive programs.

We propose to study the attainable speed-up through a detailed simulation of a computer system which directly evaluates recursive programs. The structure of the system is similar to the data flow architectures of Dennis, [Dec, 1974; May, 1975] Rumbaugh [1977], and Gostelow and Arvind [1976]. Among the results of this simulation will be data on the speed-up attainable using such architectures on various classes of problems (for example, solving systems of equations, differential equations, sorting, searching, etc.) as a function of both the number of inputs (problem complexity) and the number of microprocessors in the computer system (system complexity). As a step towards a comprehensive evaluation of data flow architectures, the results of such a simulation would be valuable to scientists doing research in data flow architectures specifically as well as to those interested in other aspects and viewpoints in the design of parallel systems.

We intend to make the simulation extensible to other related architectures. In particular, the system we will study deals with the problem of a central memory accessible by many microprocessors in an unrealistic way. (It assumes memory conflicts only occur when more than one microprocessor attempts to access the same word in memory during the same memory cycle - that is, single-word memory-banks.) In the future, we want to be able to extend our simulation to handle other hypotheses about the organization of central memory.

If results warrant it, we may be able to implement a data flow computer system using network microprocessor facilities in our laboratory at Colorado State University. (However, at this writing, funding for these facilities is still pending.)

As a first step towards an eventual simulation, we will describe a machine for parallel evaluation of recursive programs.

1.2 Underlying ideas. A safe computation rule for a program is one which guarantees completion of the computation if completion is possible at all. There are several well-known safe computation rules for recursive programs [Manna, 1974]. Many of them (e.g. full substitution) automatically provide a scheduling algorithm for evaluating subcomputations in parallel. Our simulation will quantitatively estimate the decrease in time required for execution of recursive programs when a number of processors are available to perform subcomputations simultaneously. Specifically, we will chart the percentage of speed-up obtainable by parallel evaluation of recursive programs as a function of the number of processors available. There will be a different chart for every program but by choosing the programs over a wide range of typical computations, we will present a general picture of the average speed-up attainable for different types of computations.

We want to emphasize that we are interested primarily in the speed-up attainable without regard to memory problems. There is no doubt that these parallel evaluation schemes for recursive programs will require more memory for execution than current evaluation schemes. (It is difficult to say how much more, but that figure will be another result of the study.) Further, to achieve the greatest increase in speed we will use a memory in which the words are banked individually. That is, we will assume that, in at least part of the available memory, no addressing conflicts will occur unless two processors simultaneously attempt to address the same word. Such a multi-port memory would be exhorbitantly expensive with current technology (indeed, the trend is toward larger banks, not smaller ones), but future technologies may solve this problem. In any case, we are interested in the speed-up attainable under the best memory conditions. If programs run significantly faster with these evaluation schemes, then we can attack the memory problems.

## 1.3 Relationship to previous work.

Work in parallel processing has been given a tremendous impetus by recent improvements in LSI technology. CPU's are no longer the most expensive components of computing systems. On the contrary, they are among the cheapest. It is clear that a new strategy for designing computing systems is called for, and there are many avenues which should be investigated. All researchers in the field recognize this need.

The avenue we are exploring is closely related to current research on data flow architectures [Dennis, 1974; Rumbaugh, 1977; Gostelow and Arvind 1976]. In fact the results of our study could be interpreted as an evaluation of one type of data flow architecture. However, our proposed research differs from current work on data flow architectures in two important ways. First, our machine is designed directly around an existing and well studied programming language, LISP 1.5. This avoids the pitfalls of designing new programming languages for new machines. In addition, the use of LISP avoids requiring the programmer to explicitly discover parallelism in the computation his program implements. This could be an important advantage [Glushkov, et al, 1974]. Other efforts in this general direction are Weng's design of a stream-oriented data flow language [Weng, 1975], and the important work of Kuck, Kogge and others in the translation of FORTRAN and ALGOL-like programs into programs for computer systems with a capacity for a high degree of parallelism [Kuck, et al, 1947; Kogge, 1974; Ramamoorthy, 1969; Stone, 1967]. We hope that our efforts will complement theirs and provide another set of alternatives for future researchers in the field.

A second way in which our approach differs from current work on data flow architectures is in the nature of the parallelism. Whereas most data flow schemes require a complete set of inputs before initiating a process stored at a node, subtasks in our programs are initiated as soon as they are encountered. This has important advantages in computing partial recursive functions. Not only is more concurrency allowed in some cases (namely, when a function doesn't need all its arguments to complete its computation) but it also provides a safe computation rule. In essence, our machine simulates the full substitution method of evaluation of recursive programs. This rule guarantees that a program will terminate on the widest possible range of inputs [Manna, 1974].

An important motivation for studying the execution characteristics of a recursive language is the feeling in the computing community that languages like pure LISP (i.e., variableless, recursive languages) encourage the production of well-designed, reliable, verifiable programs [Landin, 1966; Noonan and Pantor, 1974; Burstall, 1969; McCarthy, 1962; and Glushkov, et al, 1974]. If these perceptions are accurate, it may be possible to speed up the execution of programs (via parallelism) and increase program reliability at the same time by using recursive programming techniques. An additional advantage of basing our system on LISP is the existence of a large number of programs already written which may be used in gathering statistics on the speed-up attainable with our computer system. Our primary goal is to help establish the advantages of certain architectures for easily programmable computers capable of extensive parallel processing.

## 2. Multiprocessor Architecture

There are four principal components of the computer system, (1) a supervisor which maintains the list of available processors and assigns processors to tasks, (2) a collection of independent, identical processors, (3) a queue of tasks waiting to be performed (first-in, first-out), and (4) a central memory.

2.1. The supervisor. The supervisor is responsible for monitoring the task queue. If a task is waiting to be processed and a processor is available, the supervisor removes the task from the queue, removes the processor from the roster of available processors, and assigns the task to the processor. The processors themselves run asynchronously.

2.2 The processors. The computer may have any number of processor components, all of which are identical. Each processor has the following characteristics.

1. It can perform any system function.
2. It can correctly scan any compiled program.
3. It can access central memory. The four modes of memory access are:
   (a) Store value at given address,
   (b) Fetch value from given address,
   (c) Pull an address off of the free-space list (garbage allocation).
   (d) Return an address to the free-space list (garbage collection).
4. It has a local queue to keep its place in scanning compiled programs.
5. It has a small, local, random access memory.

2.3. The task queue. During any single time slice the supervisor may remove at most one task from the queue. In addition, during the same time slice, one or more processors may attempt to place a task on the queue. Only one addition to the queue may be made during a single time slice. To resolve conflicts, processors are numbered sequentially, the processors with the lowest number having the highest priority in placing tasks on the queue. Thus, conflicts are resolved in favor of the processor with the lowest number.

2.4. The central memory. The words in central memory are organized as single-word banks. Memory conflicts only arise when two processors attempt to gain access to the same word in the same time slice. As with the task queue, conflicts are resolved in favor of the processor with the lowest number. Because of the single-word banks, the memory and its addressing hardware is the most complicated component of the entire system. This is typical of multiprocessor systems of this type [Glushkov, 1974]. Reducing the complexity by increasing the size of the banks is the obvious solution, but that would obviously degrade performance. Our goal is to see what kind of performance is possible with a complex central memory before attacking the problem of reducing the memory's complexity.

Another potential memory conflict arises when two or more processors attempt to pull (or put) an address from (or onto) the free-space list. To alleviate these conflicts each processor is allocated its own free-space list by the supervisor. If a processor's free-space list becomes empty, the supervisor is notified and allocates additional free-space to the processor. Again, we assume the best of all plausible worlds with respect to the memory structure of our computer system.

## 3. Programs for the Computer System

3.1 Source programs. Source programs will be written in a subset of LISP 1.5 [McCarthy et al, 1965]. A program will be a sequence of function definitions followed by a sequence of function references. To write user-defined functions, we will use the pseudo-function DEFINE. Control within a function will be provided by the special form COND and by function references nested inside a function definition (including recursive references). There will be no other control mechanisms. It would be easy to add compatible control structures like AND and OR. However, assignments and sequential controls will not be added because the use of variables violates one of our basic assumptions (i.e., absence of side-effects).

Because the source programs will be compiled, rather than interpreted, we do not allow the function EVAL. This avoids calling in the compiler at execution time. Similarly QUOTE is only an instruction to the compiler (telling it to set up a data item in memory) and is not a system function.

To summarize, source programs will be written in a subset of LISP 1.5. The subset includes the pseudo-function DEFINE, and the special form COND. The special forms LAMBDA and QUOTE may only be used in trivial ways LAMBDA in DEFINE clauses and QUOTE for describing data to the compiler. For convenience in writing programs, the special form LIST will also be implemented and translated by the compiler into a series of references to CONS.

3.2 System functions. System functions may be thought of as machine language instructions. Each processor can directly execute a system function, given legitimate arguments.

We will write the simulator so that the set of system function can be easily changed. In this way we will be able to study the effectiveness of different collections of elementary operations.

Our current plan is to start with the set of elementary function shown in the Table of System Functions. They are taken directly from LISP 1.5. Other system functions can be added without difficulty if it seems warranted, but the ones listed here are sufficient in a theoretical sense.

Table of System Functions

| Function | Arguments | Value | Timing |
|---|---|---|---|
| CAR | non-empty list | first element of arg | 1 |
| CDR | non-empty list | arg with list element deleted | 1 |
| CONS | list or atom, list | 2nd arg with 1st arg inserted at beginning | 1 |
| NULL | list or atom | true iff arg is the empty list, else false | 1 |
| NOT* | true or false | false or true | 1 |
| ATOM | list or atom | true iff arg is an atom, else false | 1 |
| EQ | non-numeric atom or list non-numeric atom or list | true iff both arg same the same atom, else false | 1 |
| + | numeric atom, numeric atom | sum of args | 20** |
| - | numeric atom, numeric atom | arg1 minus arg 2 | 20 |
| * | numeric atom, numeric atom | arg1 times arg 2 | 100 |
| ÷ | numeric atom, numeric atom | arg1 divided by arg 2 | 300 |
| < | | true iff indicated | 20 |
| < = | numeric atom, | numeric relationship | 20 |
| = | numeric atom | holds | 20 |
| < > | | | 20 |
| > | | | |
| >= | | | 20 |
| ENTIER | numeric atom | arg converted to integer by truncation | 20 |
| LOG | numeric atom | natural logarithm of arg | 1000 |
| EXP | numeric atom | exponential of arg | 1000 |

* The atoms true and false are special system constants denoted T and F in source programs. The empty list will be treated as non-atomic. Therefore, NOT and NULL are different functions.

** Since timing in the simulation will be table driven, these timing figures can be easily changed. In this table the timing for numerical operations reflects our assumption that component processors will be microprocessors with a small word size and no built-in floating point operations. Numerical operations, if implemented in software or firmware would be relatively slow as shown here. With more sophisticated component processors, these figures would be substantially reduced.

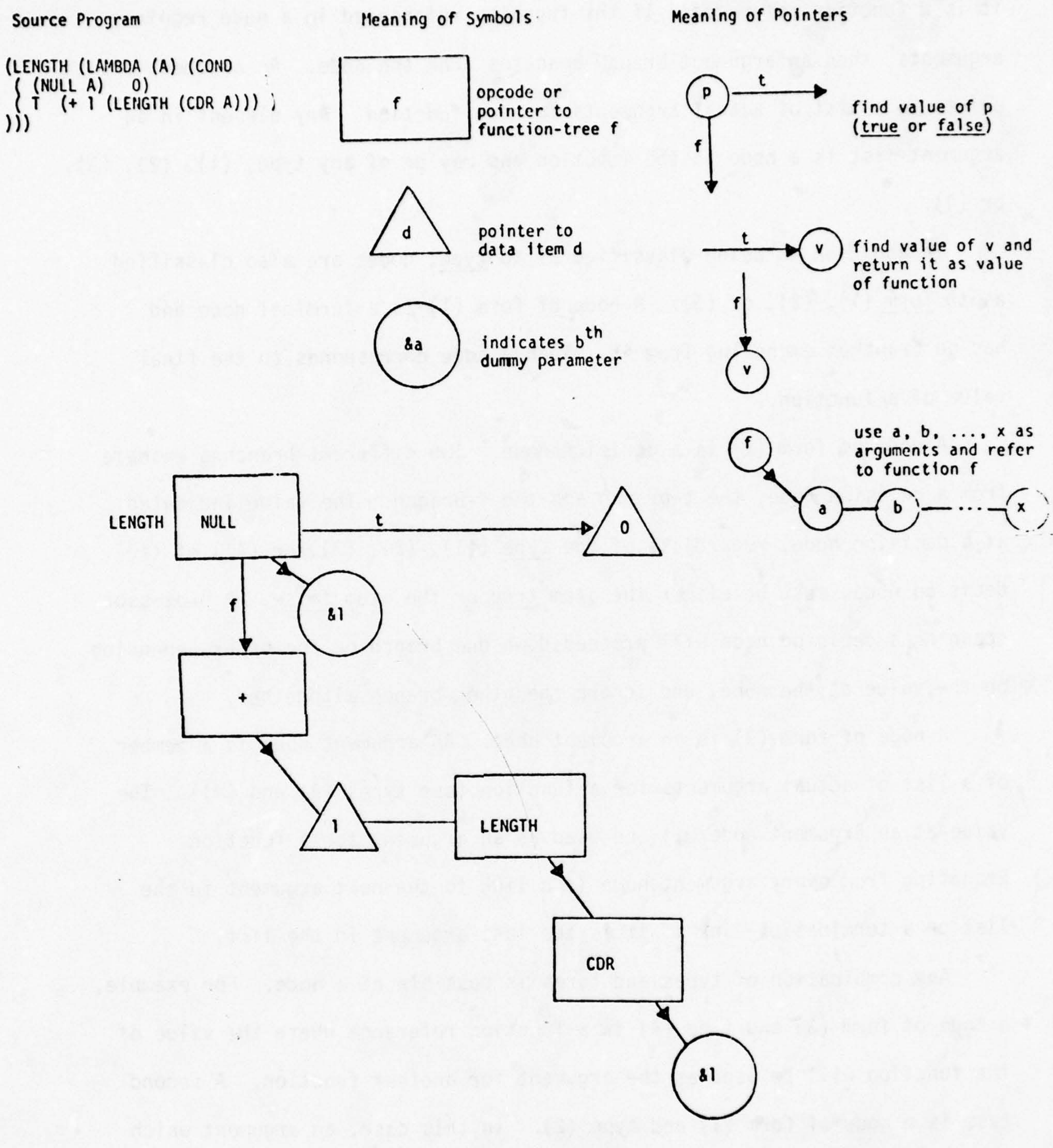
3.3 Compiled programs. At execution time all user programs will be represented as tree structures. Function references will be by numeric address of the code, data references by numeric address of the storage location, and argument references by indirection through an argument table stored in the local memory of the processor performing the function. The compiler will translate source programs into the appropriate tree structure, (see Fig. 1) replacing all symbolic references to functions, parameters, and data by numeric addresses. Each address in the compiled code will be flagged as to type: function reference, data reference, or parameter reference.

By compiling programs in this way, we delegate the need for associative memory to the compiler, and it can be handled there in the traditional way (i.e., by keeping a symbol table). Avoiding the need for fast associative memory at execution time is an important way to reduce the cost of this system. Other researchers [Glushkov, 1974] have found associative memory necessary in their designs of computer systems of this type.

Because the compiled programs are tree structures and because operations have no side effects, all code is reentrant. In fact, the code for a function may be entered at any node, thus allowing partially executed functions to be resumed at the point of departure without rescanning any previously executed code.

## 4. Scanning Compiled Programs

4.1. Classification of nodes in function trees. A compiled program is a collection of function trees. To scan a function tree is to trace through its nodes in the appropriate order. A node in a function tree may be (1) a pointer to a data time (a value node), (2) an index into an argument list (a dummy-parameter node), (3) the operation code for a system function, (4) a pointer to a function tree. If a node is of type (3) or (4), then


Source Program
Meaning of Symbols
Meaning of Pointers
f
opcode or pointer to function-tree f
d
pointer to data item d
&a
indicates b th dummy parameter
p
t
f
find value of p (true or false)
t
v
f
v
find value of v and return it as value of function
f
a
b
...
x
use a, b, ..., x as arguments and refer to function f
LENGTH
NULL
t
0
f
&1
+
1
LENGTH
CDR
&1

```
(LENGTH (LAMBDA (A) (COND
 ( (NULL A)   0)
 ( T  (+ 1 (LENGTH (CDR A))) )
)))
```

Compilation of a Source Program

Figure 1

it is a function reference. If the function referenced in a node requires arguments, then an argument-branch emanates from the node. An argument-branch points to a list of actual arguments for the function. Any element in an argument list is a node in the function and may be of any type, (1), (2), (3), or (4).

In addition to being classified as to _type_, nodes are also classified as to _form_ (1), (2), or (3). A node of form (1) is a terminal node and has no branches emanating from it. Such a node corresponds to the final value of a function.

A node of form (2) is a decision node. Two different branches emanate from a decision node, the t-branch and the f-branch. The value indicated at a decision node, regardless of the _type_ ((1), (2), (3), or (4)) of the decision node, must be either the atom _true_ or the atom _false_. A processor scanning a decision node will proceed down one branch or the other depending on the value at the node, and ignore the other branch altogether.

A node of form (3) is an argument node. An argument node is a member of a list of actual arguments for a function (see types (3) and (4)). The value at an argument node will be used as an argument for a function. Emanating from every argument node is a link to the next argument in the list or a termination-link if it is the last argument in the list.

Any combination of types and forms is possible at a node. For example, a node of form (3) and type (4) is a function reference where the value of the function will be used as the argument for another function. A second case is a node of form (1) and type (2). In this case, an argument which has been passed into a function (pointed to by a dummy parameter) is being used as the value of the function. It should be noted that this division of types and forms prohibits the use of CONDitional expressions as arguments in function references. This is not a serious restriction, however. Programs which seem to need such a construction are better written with the COND-clause replaced by a call to a helping function.

4.2 The Task Queue. Entries on the task queue are either class I entries (function evaluations) or class II entries (decision nodes awaiting predicate values).

A Class I entry is a record with the following components:

- o function-tree address or system-function op-code
- o argument table address
- o destination address

A Class II entry is a record with the following components:

- o predicate value address
- o T-branch address
- o F-branch address
- o argument table address
- o destination address

When a processor pulls a class I task from the queue, it sets its program counter (PC) to the indicated function-tree address or op-code and sets its argument table address register and destination address register to the values in the task queue entry. Then it goes into function-scan mode.

When a processor pulls a task of class II off the queue, it examines the contents in the indicated predicate value address. If the predicate value is undefined, the processor puts the class II task back on the queue and releases. If the predicate value is defined, the processor sets its PC to the node indicated in either the T-branch or F-branch of the task (depending on the predicate value, of course). It also sets its destination address register and argument table address register to the values indicated in the task. Then the processor goes into function-scan mode.

4.3 Function Scan Mode. The following rules are used by processors in scanning functions.

4.3.1 The value of a node may be obtained directly if one of the following conditions holds;

(a) The node is of type 1 (data value)

(b) The node is of type 2 (dummy parameter) and the memory word pointed to indirectly through the argument table has been defined (again the value of the node is of the value of the word).

4.3.2 If a node of form 1 (terminal) is encountered and its value may be obtained directly, place a pointer to its value in the memory word indicated for the result when the function reference was pulled off the global task queue and release.

4.3.3 If a node of form 1 (terminal) and type 2 (dummy parameter) is encountered and its value cannot be obtained directly, the processor sets the result destination to an indirect address pointing to the address indicated by the node and releases.

4.3.4 If a node is of form 1 (terminal) and type 3 (system function) or type 4 (user functions), its value cannot be obtained directly and the processor goes into argument scan mode as described below. Note that the destination address for the value of the node is currently set to the destination address of the function being scanned.

4.3.5 If a node is of form 2 (decision node) and type 3 (system function) or type 4 (user function), the processor will generate a class II task to put on the global task queue. This involves grabbing a word of memory (call it P) for the predicate value part of the entry. Then get the pointers for the two branches of the decision node (T and F). Finally retrieve the original destination address and the argument list from registers. Once this is done, this class II task is put on the task queue. Now start to create a class I task with destination-address register set to P. Obtain the op-code or function address. Then go into argument scan mode to create the argument list.

4.4. <u>Argument Scan Mode.</u> The following rules are followed by processors in <u>argument scan mode</u>. In this mode the current node must be of type 3 (system functions) or type 4 (user functions).

4.4.1 Save the current position by storing the current destination address and the address of the current node on the processor's <u>local</u> queue.

4.4.2 Scan the nodes in the argument list of the current node in sequence.

(a) For a node being of type 1 (data value) or type 2 (dummy parameter), obtain its address and place it in the argument table being built in local memory.

(b) For a node of type 3 (system function) or type 4 (user function), get a word of main memory (its value will be currently undefined) and place its address in the argument table being built. The word is where the function's answer will be stored later, so the argument table entry will eventually point to an answer. In addition, put an entry on the local queue consisting of a pointer to this node and the destination address.

4.4.3 Back up and evaluate function by removing the first entry from the local queue. If it points to a node of type (4) (function-tree), copy the pointer to the function-tree from the node and <u>put</u> it on the (global) task queue along with the current destination address and a pointer to the argument table just prepared. On the other hand, if the entry just removed from the local queue points to a node of type (3) (system function), try to execute the system function. If the execution is successful, place its value in the current destination address. If it isn't successful, <u>put</u> its op-code, the argument table, and the destination address on the (global) task queue.

4.4.4 Evaluate the functions encountered in the argument scan. If the local queue is empty, release. Otherwise, remove the first element. That is, set the PC to the node address in this element, and set the destination address register to the destination address in this element. Then proceed from step 1 in these argument-scanning instructions.

## 5. Conclusion

We have discussed a way to organize a large number of processors into one computer system. The system can be viewed as a single-program multiprocessor network with shared memory and a multi-instruction, multi-data (MIMD) stream. The system is easily expandable--new processors can be added (or deleted) without modifying the overall structure. Scheduling is relatively simple because programs for the computer system consist of a number of asynchronous subtasks. Our immediate goals in future work are to develop a simulator for the system and measure speedups attainable as a function of the number of processors available and the tasks being performed. Secondary goals include studying the use of memory by the processors as they perform tasks in an attempt to discover a way to further reduce the complexity of the memory structure without significantly degrading performance.

## References

Burstall, R. M. Proving properties of programs by structural induction, Computer J., Vol. 9, (Feb. 1969), pp. 41-48.

Chamberlin, D. D. The "single-assignment" approach to parallel processing, Proc. 1971 FJCC, AFIPS, pp. 263-269.

Dennis, J. B. First version of a data flow procedure language, MAC TM 61, Project MAC, MIT, (May 1975).

Dennis, J. B. and Misunas, D. P. A preliminary architecture for a basic data flow processor, The 2nd Annual Symposium on Comp. Arch., Houston, (Jan. 1975), (ACM-SIGARCH, Vol. 3, (Dec. 1974), pp. 126-132.

Glushkov, V. M., Ignatyev, V. A., Myasnikov, V. A., and Torgashev, V. A. Recursive machines and computing technology, Proc. IFIP 74, North-Holland, Amsterdam, (1974), pp. 65-70.

Gostelow, K. P., and Arvind. A computer capable of exchanging processing elements of time, Tech. Rpt. #77, Univ. of Calif., Irvind, (Jan. 1976).

Holland, J. A universal computer capable of executing an arbitrary number of sub-programs simultaneously, Proc. 1959 EJCC.

Kogge, P. M. Parallel solution of recurrence problems, IBM J. Res. Develop., Vol. 18, (1974), pp. 138-148.

Kuck, D. et al. Measurements of parallelism in ordinary FORTRAN programs, IEEE Computer, (Jan. 1974), pp. 37-46.

Landin, P. J. The next 700 programming languages, Comm. ACM, Vol. 9, (Mar. 1966), pp. 157-166.

Manna, Z. Mathematical Theory of Computation, McGraw-Hill, New York, (1974).

McCarthy, J., et al. LISP 1.5 Programmer's Manual, MIT Press, Cambridge, (1965).

McCarthy, J. Toward a mathematical science of computation, Proc. IFIP, Conf. 1962, pp. 21-27.

Noonan, R. E., and Pantor, D. J. Structured recursive programming, Lecture Notes in C.S., 19, Goos and Hartman, eds., Springer-Verlag, New York, (1974).

Ramamoorthy, C. V. and Gonzalez, M. J. A survey of techniques for recognizing parallel processable streams in computer programs, Proc. 1969 FJCC, AFIPS, pp. 1-15.

Rumbaugh, J. A data flow multiprocessor, IEEE Trans. on Computers C-26, 2, (Feb. 1977).

Stone, H. S. One-pass compilation of arithmetic expressions for a parallel processor, Comm. ACM, Vol. 10, (Apr. 1967), pp. 220-223.

Urschler, G. The transformation of flow diagrams into maximally parallel forms, 1973 Sagamore Conf. on Parallel Processing, pp. 38-46.

Weng, Kung-Song, Stream-oriented computation in recursive data flow schemes, MAC TM 68, Project MAC, MIT, (Oct. 1975).

# TECHNEC,
# A Network Computer for Distributed Task Control

Wing Huen, Peter Greene, Ronald Hochsprung and Ossama El-Dessouki

DEPARTMENT OF COMPUTER SCIENCE

ILLINOIS INSTITUTE OF TECHNOLOGY

Chicago, Illinois 60616

This work was supported by National Science Foundation Under Grant MCS 76 - 01310 with Peter Greene as Principal Investigator.

## ACKNOWLEDGEMENT

Several members of the Department of Computer Science at Illinois Institute of Technology have made significant contributions to this project. We would like to thank Dr. Robert Tobey for his encouragement throughout the project and leadership in organizing the implementation effort. Thomas Christopher participated in an initial stage of design and served as project manager. The implementation of hardware and software was carried out by an extraordinarily capable group which consists of O. El-Dessouki, F. El-Wailly, A. Hazra, K. Sista and W. Wojciechowski.

## Abstract

This paper reports investigations on new ways to use a ring network of microcomputers in which the constituent computers are loosely coupled. A single program is to be partitioned and run as parallel distributed tasks on the network. The design considerations, hardware configuration, software facilities, and communication protocol are described. Some processes, such as compilation, simulation, and process control, which are traditionally performed on uniprocessors are considered for the network computer.

## I. Overview

This paper presents the design considerations, outlines the organization of TECHNEC, the Illinois Institute of TECHnology NEtwork Computer, and two of the initial ways in which it will be used. We call it a "network computer", rather than a "computer network", because we are not thinking of running simultaneously a number of jobs that share network resources but rather a single job at a time, which is distributed over a collection of microprocessors.

The paper is organized as follows:

## II. Design Decisions

There is a general sentiment that it is feasible and cost effective to construct powerful computer systems with multiple microprocessors.[1,2,4] However, a number of hardware and software issues concerning interconnection strategy, communication, resource allocation, and performance are still unclear.[3,5] We shall address some of these design problems in this section before the TECHNEC is described.

We shall use the term "interconnected computer systems" to designate computer structures composed of multiple processors, memories, and I/O devices. Interconnected computer systems thus include multiprocessor systems, in which each processor may access all memories and I/O devices, and computer networks, in which each computer accesses only its own private memory and communicates with other computers with messages.

The construction of an interconnected computer system is a series of design decisions. Our considerations for the TECHNEC are presented below.

### 1. Mode of Use

Before designing a system one must consider a strategy to exploit the system, for this will affect many of the structural features of the system. Three strategies for exploiting multiprocessor systems[5] and networks are:

a) Single Special Task Environment. The system is designed for a specific task. An outstanding example is the PLURIBUS IMP[6], which is connected to service one single special function, namely message switching for the ARPA net. Since the application is specific, the hardware and software of the system can be structured optimally.

b) Standard User Environment. Multiprocessor systems such as HIS 6180, IBM 370/168 MP, CDC6600, and Burroughs 6700 provide a user environment quite similar to that of single processor systems. The only difference is that they achieve higher throughput, reliability, computing power, and economies of scale.

Computer networks such as the DCS[7,8,9] and the SPIDER[10] enable resources sharing, so that a user at one computer may request services available on other computers in the network. Programs may be relocated by the network dynamically from processor to processor to achieve balanced use of resources. To the user, the computer network appears to be a single facility with a wider range and selection of services than a minicomputer.

c) Multiple Specialized Application Systems. In this strategy, a number of specialized systems are realized on an interconnected computer system. Two examples are the C.mmp[11] and Computer Modules[3]. Applications such as speech analysis and visual recognition may be run simultaneously or singly on the C.mmp, each application occupying one or more processors. The architecture and communication path are fixed, although the design is modular enough that configurations may vary in the number of processors, the size of the switch, and memory size.

The designers of the Computer Modules went further to permit the sophisticated user to reorganize the hardware structure and the communication pattern for a specific task.

The TECHNEC project aims to develop multiple specialized applications on an interconnected computer system with a fixed architecture as in C.mmp. The system is dedicated to the parallel execution of a single application at a time, such as continuous as well as discrete simulations, and complex process control--tasks which are traditionally executed on single processor systems.

## 2. System Characteristics

The interconnected computer system should have the following system characteristics:

a) Modularity. The capacity of the system should be incrementally changeable. There are two measures of modularity: cost modularity and place modularity[12]. A system is cost modular if the incremental cost of adding an element such as a processor is simply the cost of the element. Place modularity refers to the amount of freedom with which an incremental element can be inserted in the structure. The interconnected computer system should be highly modular with respect to these two measures.

b) Logical Complexity. Since our effort is mainly exploratory, the hardware, software, and communication pattern should be logically simple to comprehend and implement.

c) Distributed Capabilities. The architecture should lend itself easily to distributed processing. A single computational task is to be partitioned to run in a distributed fashion on the interconnected computer system.

d) Failure Effect and Failures Reconfigurability. The effect of failure of a processor or a communication path on the whole system should be minimal. It should be easy to reconfigure the system after failure. One approach is to have interchangeable components for ease of maintenance.

## 3. Interconnection

The taxonomy of interconnection by Anderson[12] is instrumental in formulating design problems for the TECHNEC. Ten interconnection designs were identified in the taxonomy. The first choice in interconnection strategy is between direct and indirect transmission of messages from source to destination. In direct communication, a path in the form of a link, a bus, or a memory connects two processors. In indirect communication, an intervenor exists to alter the messages (e.g., address transformation) or to route the message onto one of a number of alternative output paths. For example, in the SPIDER, which is a loop network with a central switch, two processors intending to engage in communication must first report to the central switch. Each message is sent to the central switch with the sender's identification. The central switch then routes the message to the receiver with the receiver's identification.

Other examples of intervenors in indirect communication are the central switching node in a star network and the memory mapping functions across buses in Computer Modules, which are interconnected bus systems. Accessing a memory on a distant bus in Computer Modules takes variable time which depends on the number of intermediate mappings. Moreover, interconnected bus systems are susceptible to deadlocks[13].

Because of the relative complexity in indirect communication, we have decided to make use of direct communication.

The second decision to make with respect to interconnection involves the choice of <u>shared</u> or <u>dedicated</u> message transfer paths. A path is considered to be shared if it is accessible from more than two points at the same time.

Two techniques provide a shared transfer path: global bus or multiprocessor. Access to the global bus is shared among the processors by some arbitration scheme. Messages are sent from the source processor onto the bus, to be recognized and accepted by the proper destination. It is simple to add additional processors, but cost modularity and place modularity of the bus are poor. It is impossible to increase the bandwidth, and reduction of contention often necessitates multiple buses or bus replacement. Failure of individual processors does not cause serious impact, because multiple processors are available. It is simple to amputate faulty processors but bus failure is disastrous.

In multiprocessor architectures, two or more processors share a common physical memory space. As a side effect, this common memory is used for message communication. The simplest common memory access mechanism is a bus. Implementations more commonly use multiple memory buses or very expensive central switches. It has been found that the performance of multiprocessor systems increased more slowly as the number of processors increased because of contention for memory bandwidth. It appears that effective use of such systems requires fundamental understanding of program partitioning. The cost of the central switch is also out of our reach.

Two implementation alternatives are available in the dedicated path approach: complete interconnection and the ring. In the complete interconnection approach, each processor is connected directly to every other processor, and messages are exchanged between any pair of processors along the dedicated link. This organization has the advantages of logical simplicity and the capability of accommodating any communication pattern among processors. Cost-modularity, however, is very poor. Addition of a processor to an n-processor system requires the addition of n paths, and all the existing processors must be made ready to handle incoming messages from the incremental processor.

We are left with one candidate structure: the ring, which provides a single direct, dedicated communication path between adjacent pairs of processors. Our decisions to adopt the ring structure are as follows:

a) Modularity. The cost-modularity and place-modularity of the ring are very good. The cost of adding a node is the cost of a processor and the associated portion of communication path.

b) Logical Complexity. The logical complexity of the ring is very low. Each node communicates with only its predecessor and its successor. The hardware and software for each node can be made completely identical.

c) Distributed Processing. Although programs are relocatable across processors in the DCS or SPIDER, the execution of a program is essentially localized at one specific location. Distributed processing is straightforward in multiprocessor systems via the common memory, but protection and system integrity are serious problems.

The concept of a network computer intrigues us, namely to run a program in a distributed manner on a network of loosely coupled computers, which behaves as a powerful computer system. The loose coupling isolates portions of a program and enhances system integrity. This concept may be advantageously employed in future systems in which more intelligence is added to peripheral processors. The price paid is the relative slow communication rate between processors. Partitioning of program and data becomes an important problem in using such a system effectively.

The ring provides a straightforward environment for investigations on pipelined processing.

d) Failure Effect and Failure Reconfigurability. Failure effect of a processor is minimal, as in the global bus approach, but the failure effect of the ring communication path is not as bad because the communication path can be implemented with microprocessors. The probability of failure of all the processors is rather small.

Traditionally, reconfiguration after failure has not been performed on ring networks such as DCS and SPIDER, which are constructed to link geographically dispersed computer systems for file and resource sharing. But in our intended use, the ring can be reconfigured easily to a smaller ring by removing failed processors or communication microprocessors.

e) Performance Improvement. Traditional ring networks use unidirectional, bit-serial interfaces. Byte-parallel interfaces are available for implementation of communication path with microprocessors. Moreover, the bandwidth can be improved by refinement of message protocol and message buffering at communication processors. Bidirectional message communication may also be attempted.

## 4. Communication

The design of a message protocol for a ring network needs to consider the following factors: direction of communication, number of receivers per message, message length, coupling between sender and receiver, and message buffering.

Unidirectional communication was chosen because of the complexity of bidirectional communication.

One-to-many (broadcast) mode is desirable as a control mechanism to synchronize parallel processes in applications such as simulation and demon control.[15] Point-to-point communication is the basic mode of correspondence between processes.

The message size is often dependent on the nature of an application. Variable message size is appropriate since fixed size packets are wasteful for short messages which are the majority and an elaborate disassembly-assembly mechanism is necessary to handle messages longer than a packet.

The DCS identifies processes by unique names. A message is addressed to a process, typically a compiler or an editor, by name rather than location because processes are moved dynamically within the network to achieve balanced loading. A requesting process does not and should not know the location of the receiver process. In the TECHNEC, communication overhead is incurred, as a result of program partitioning, by allocating interdependent processes to separate computers. The communication pattern is predetermined before execution. The processes are nameless but some form of identification is essential to identify the logical communication path. Virtual channels are assigned for each logical communication path to delay process-processor binding. Moreover, the concept of virtual channels extends easily for the broadcast mode.

The TECHNEC also differs from the DCS in message buffering. In the latter, outgoing messages are queued in an output queue waiting to get on the ring. A message, on arriving at the destination processor, is entered into the input queue of the receiver process. On the TECHNEC, a sender must have enough space for all the outgoing messages and must keep a copy of a message until the message has made a round trip around the ring. This policy effectively prevents a process from using up all the free space at a receiver.

## 5. Deadlocks

Deadlocks can occur when multiple messages, each occupying a part of the communication path, attempt to access the part occupied by another message.[13] The TECHNEC is basically a Newhall[14] loop that permits at most one message to be transmitted at a time.

## 6. Resource Allocation

The major resources of the network are processors and local memories. Since each processor accesses exclusively its local memory, the two resources are one. The resource allocation problem becomes one of program partitioning so that the overall execution time of the program is minimal.

## 7. Process to Processor Binding

A process will be bound to a processor and physical memory to simplify network software and memory management. A scheduler is necessary for multiple processes coexisting within the same processor. The use of virtual channels instead of processor identification and locations delays the binding and facilitates processor assignment.

## III. Network Architectures

As was mentioned earlier, TECHNEC is a ring of 12 nodes (Fig. 1.) A node is composed of a Ring Interface Unit (RIU) and a MicroProcessing Unit (MPU). The RIUs, linked by I/O ports, are responsible for message communication between nodes.

Both the MPU and the RIU are implemented by microcomputers for economic reasons. Also one of our main objectives in this project is to investigate ways of utilizing a network of inexpensive microcomputers to perform applications such as the examples sketched in Section V.

Each MPU is implemented by an LSI-11 with at least 12K words (up to 56K bytes) of RAM and floating-point hardware. LSI-11 was chosen mainly because of the considerable amount of system software support accessible to LSI-11 users and availability of floating point instructions, which are essential to our applications. Moreover, the LSI-11 assembly language is powerful enough for software development. Another factor is our familiarity with PDP-11 assembly language and its general structure. This relative ease of software development allows the research group to concentrate on investigations in distributed processing.

Each RIU is implemented by an 8-bit RCA COSMAC with 1K bytes of RAM. Each RIU provides two functions. It serves as an interface between the MPU and the ring, and also communicates with adjacent RIUs. The RCA COSMAC was chosen to implement the RIU because it is a "low end" device suitable for simple applications with limited programming needs. Being fabricated with CMOS technology, it requires very little power. It operates with a single power supply (between +3V and +12V ) and is very insensitive to noise. Four flags, which are connected directly to CPU pins, can be set or reset by external logic to control actions of the COSMAC CPU. For example the EF3 flag can be controlled by the MPU and one COSMAC instruction is needed to branch on high or low condition of the flag. The COSMAC also provides a DMA facility suitable for easy, efficient,and automatic program loading and data transfers. The COSMAC instruction repertoire, although limited, is sufficient for implementing message protocols. Moreover, since

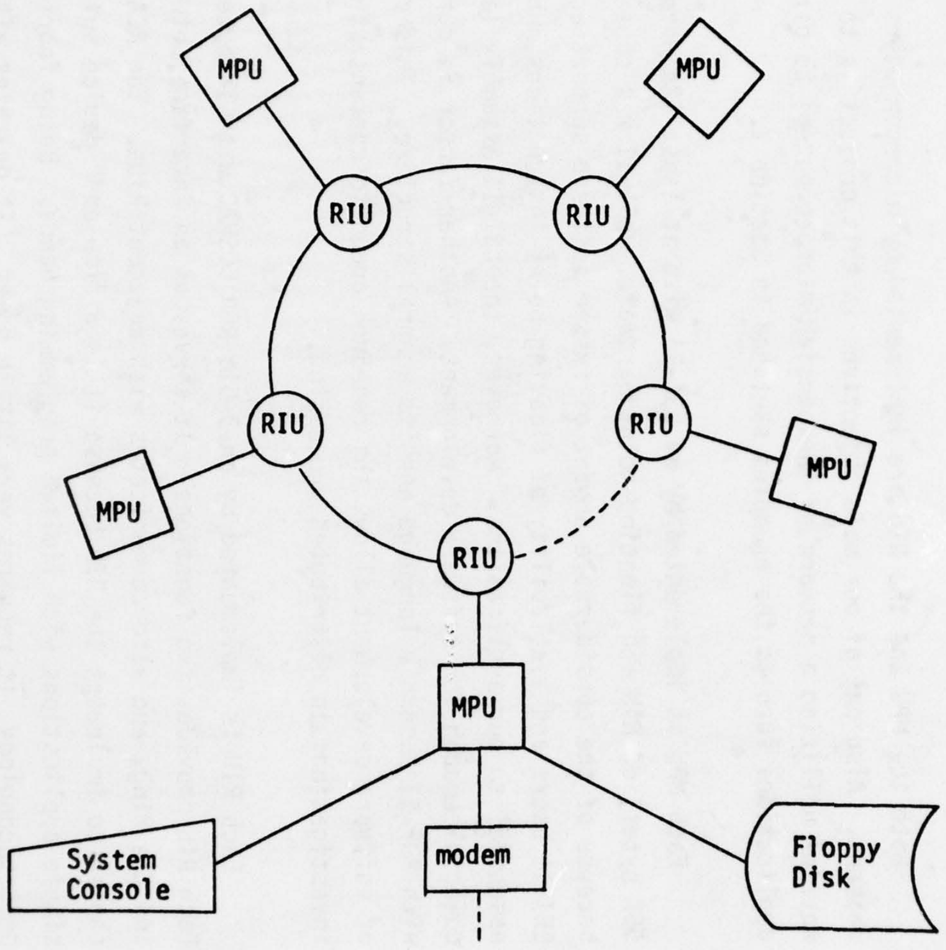


Figure 1. Structure of TECHNEC.

the RIU is transparent to the TECHNEC user, powerful programming facilities are not required. These factors made the COSMAC very attractive as a candidate in RIU implementation.

An RIU communicates over an 8-bit data bus to which two interfaces are connected: RIU - RIU interface and RIU - MPU interface (Fig. 2).

## RIU - RIU Interface

Communication between two adjacent RIUs is strictly asynchronous, byte-parallel, and unidirectional. RIU(n+1) and RIU(n) always have a master-slave relation, communicating via an I/O port that is realized by an INTEL-8212 chip. An I/O port is implemented as a memory-mapped device in the RIU's address space. The EF1 flag of RIU(n+1) and the EF2 flag of RIU(n) (Fig. 3), are connected to the INT output of the 8212 chip. In the quiescent state, both flags are set high. Each RIU continually pools its input port for data arrival from the previous RIU. When RIU(n) writes into its output port, the EF1 flag of RIU(n+1) and the EF2 flag of RIU(n) are pulled low. The EF2 flag remains low until RIU(n+1) reads from its input port. This interlock prevents RIU(n) from writing further data into its output port. The EF1 flag at the low condition indicates to RIU(n+1) that it may read. A read operation by RIU(n+1) from its input port removes the byte and sets both flags high, enabling RIU(n) to write again.

## RIU - MPU Interface

There are two types of interfaces between an RIU and the corresponding MPU:

a) Register Interface. This is a normal PDP-11 interface. There is a Control and Status Register (CSR) and a Data Buffer Register (DBR). An RIU and its corresponding MPU can communicate by setting control and status bits (operation code) in the CSR. The DBR is often used for passing addresses between MPU and RIU.

b) Direct Memory Access (DMA) Interface. This interface is controlled

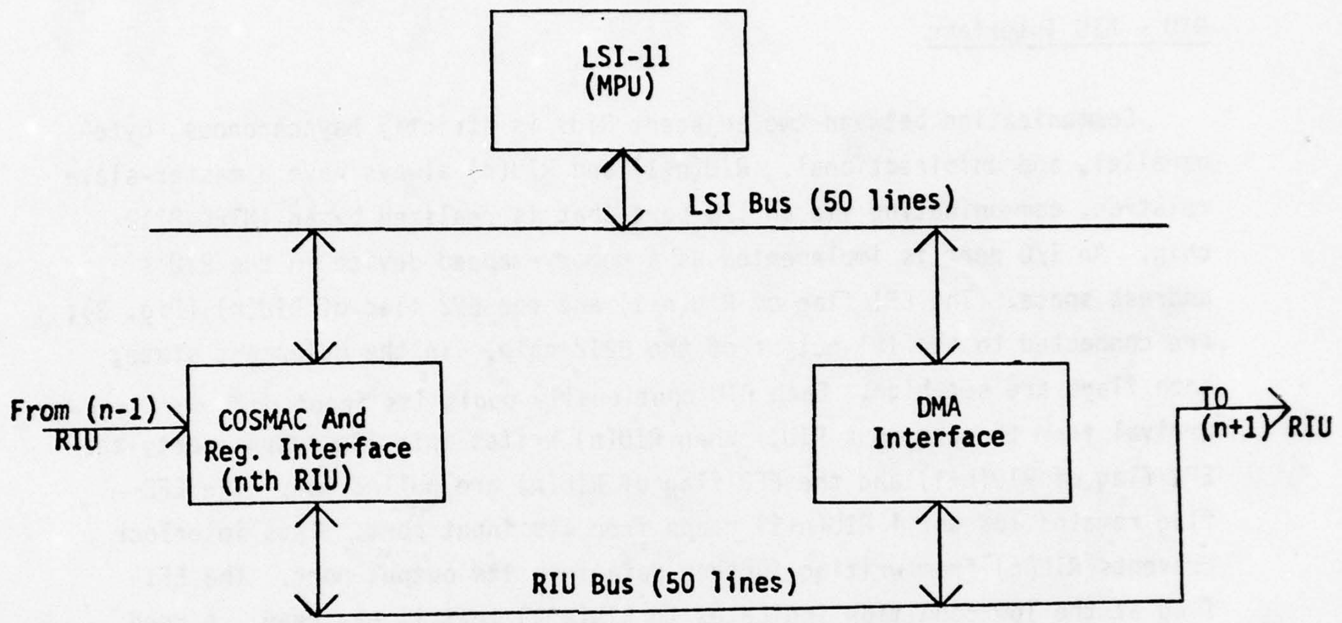
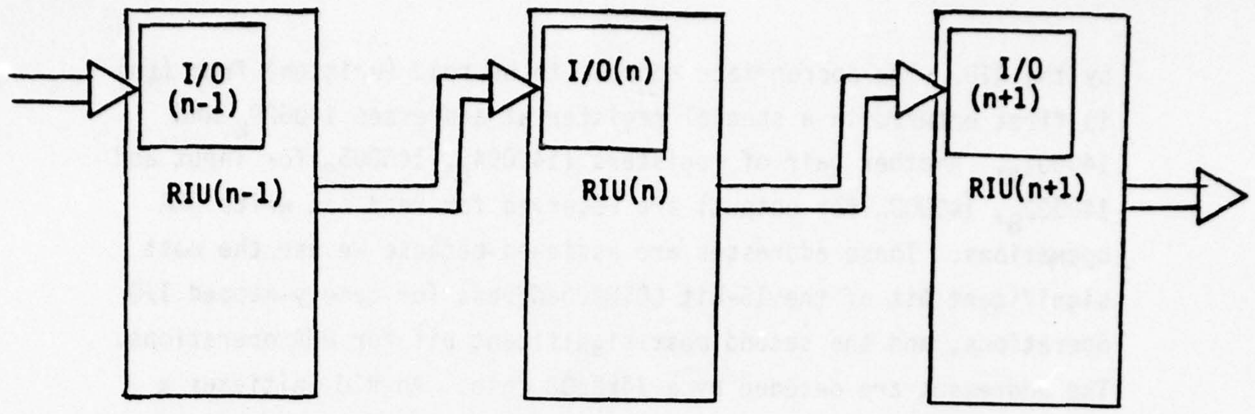


Figure-2 MPU-RIU Interface



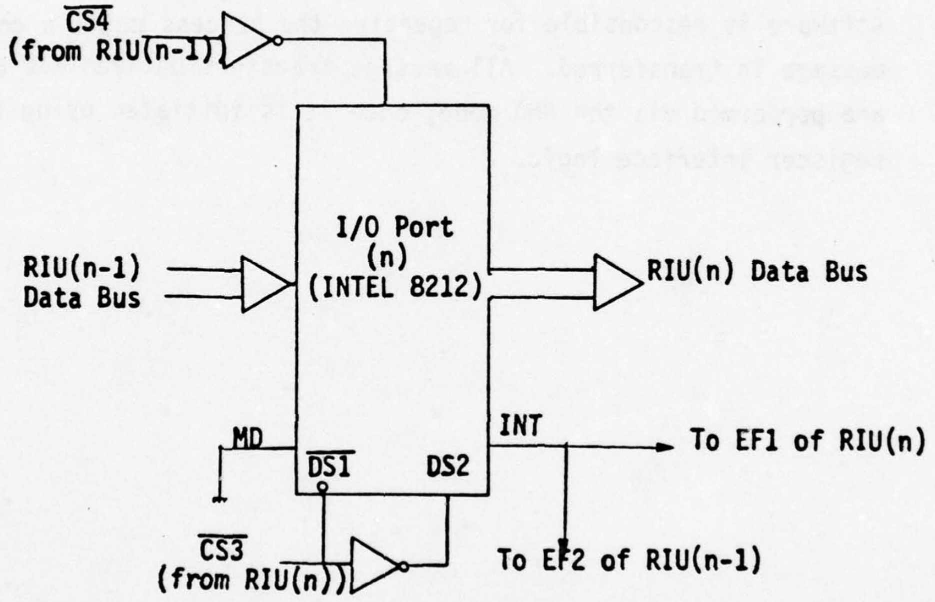

$\overline{CS4}$ (from $\overline{RIU(n-1)}$)
I/O Port (n) (INTEL 8212)
RIU(n-1) Data Bus
RIU(n) Data Bus
INT
To EF1 of RIU(n)
MD
$\overline{DS1}$
DS2
$\overline{CS3}$ (from $\overline{RIU(n)}$)
To EF2 of RIU(n-1)

Figure-3 COSMAC-COSMAC Interface

by the RIU. The appropriate address to be read (written) from (to) is first entered in a special register at addresses $140000_8$ and $140001_8$. Another pair of registers ($140004_8$, $140005_8$ for input and $140002_8$, $140003_8$ for output) are reserved for read and write DMA operations. These addresses are assigned because we use the most significant bit of the 16-bit COSMAC address for memory-mapped I/O operations, and the second most significant bit for DMA operations. The addresses are decoded by a 74LS138 chip. An RIU initiates a DMA operation by writing a special operation code (2 for interrupts, 4 for DMA-write, and 5 for DMA-read), in the memory-mapped device address $140007_8$. This operation code serves as a DMA strobe signal. Once a DMA operation is initiated, a custom-made diode matrix utilizes the DMA facility on the LSI-11 and performs the required "handshaking" with the LSI-11 bus for cycle stealing. The COSMAC software is responsible for repeating the process until a complete message is transferred. All message transfers between RIU and MPU are performed via the DMA mode, once it is initiated using the register interface logic.

## IV. Software Facilities

The most important software facility on TECHNEC is the message communication routine. There is a message manager routine (MSGMAN) in each MPU and in each RIU a software module, MSGBOY, which cooperates with the corresponding MSGMAN in message transmission and reception. Other facilities include multitasking dispatching, console communication, file management and debugging. More space will be dedicated to the description of the message communication mechanism since the other facilities are rather conventional.

### Communication

Nodes communicate using a unidirectional protocol. A message passes through each node and is removed from the ring when it is received in the node which originated it.

The design of the message protocol allows the possibility of having more than one message passing around the ring at a time; this implies a buffering capability within each node. Owing to lack of buffering, only one message is active on the ring at any time in our current implementation.

Messages are passed over a group of "virtual" channels. The system currently has no dynamic channel assignment facility, so the allocation of channels must be done by the users. Certain channels are pre-assigned for internal system functions (such as program loading and debugging operations). Each request to the message passing system specifies the channel number over which the message is to be sent or received.

Two styles of messages are supported: "point-to-point" and "broadcast" messages. The point-to-point style of message is the simpler to use. Each message request includes a "subchannel" number in addition to the channel number. It is assumed that only one task will actually be waiting for a message over the channel/subchannel number combination. This style is more frequently used by existing programs. The channel/subchannel number combination is effectively the name of a task in a point-to-point communication.

The "broadcast" message style was included to allow the one-to-many mode of communication. In broadcast usage, the sending task does not know what task(s) may be waiting for a message on the channel; indeed, there may be no tasks currently waiting. Likewise, a receiving task has no way of knowing what other tasks also received a copy of the broadcast. Also, the system does not provide any identification of the sender in the protocol, though this could be included within the data portion of the message if desired.

The system uses dynamic buffer allocation for the receipt of broadcast messages. As many tasks as wish to receive a broadcast message over a single channel are allowed by MSGMAN to share a single copy. The buffer is not freed until all such tasks indicate that they no longer need the message. Each message contains a routing field that is read and reset by receivers and is used to convey reception status information to the sender.

The message passing operations are performed by MSGBOY. It is responsible for receiving messages from the preceding RIU, interrupting its MPU to indicate the arrival of messages, obtaining data from its MPU by DMA mode for transmission, passing messages to the succeeding node,and other functions such as loading code into its MPU and dumping the contents of its MPU. The MSGBOY is invisible to the user task in the MPU.

User tasks in MPU make requests for service to MSGMAN,which maintains tables and queues indicating the state of the message passing mechanism. It also contains an entry point which services the RIU interrupt. Whenever the MSGBOY needs a decision regarding what to do, it interrupts the MPU, passing a code through the status register telling what is required. MSGMAN then examines its tables and returns a code to the RIU, informing it what to do.

When the RIU examines the CSR and finds that it should copy the message into the MPU, it will proceed without any further interaction with the MPU until the end of the message is detected. At this point it will generate another interrupt. MSGMAN will then update its tables, unblock any waiting tasks, etc.

If the RIU is not to copy the message, it simply passes the message through. In either case, it will appropriately update the routing field. When the RIU detects the "caboose" of a possible train of messages, it will generate an interrupt to inform the MPU that a message can be transmitted. MSGMAN will interrogate its tables and inform the RIU as to whether a message is to be sent or not. If yes, then the RIU will add the message to the ring. When it detects the message coming in (having passed around the ring), it removes the message (i.e., does not pass it on) and interrupts the MPU to allow MSGMAN to again update its tables.

Improvements are being considered to achieve higher communication bandwidths by adding buffering at the RIUs. Multiple messages can then be transmitted simultaneously. Additional memory to accommodate the list of channel numbers at each RIU would also enable the RIU to take a more active role in deciding on appropriate actions on a message instead of interrupting the MPU each time. Since the list of channel numbers may be changed dynamically, the capability of an MPU to interrupt its RIU is also desirable.

## SEXTECH (System Executive)

A program running on TECHNEC is to be structured as one or more tasks. On each MPU there is a resident multitasking dispatcher called SEXTECH. The SEXTECH allows multiple tasks to reside in a MPU and schedules user tasks in a simple round robin fashion.

## SUPERTECH

The system node provides utility functions. It interacts with the user terminal, floppy disks, and phone lines to other computers on campus. This node runs a collection of facilities which are jointly called SUPERTECH. One such facility (CONMAN) allows user tasks to interact with the system terminal. Another (FILMAN) allows user tasks to access files stored on the floppy disks. Yet another (DEBUG) interacts with the user (using CONMAN) to allow debugging operations to be performed over the ring. This is absolutely necessary since the user nodes have no terminals, no display lights, etc. The only way to examine the contents of memory in user nodes is via the DEBUG facility. User tasks request actions by these facilities by sending messages over special channels.

The DEBUG module provides functions such as suspension of a task, resumption of a task, modifying contents of a location, display of status, and breakpoints.

## V. Application Software

Several ongoing projects are developing software geared to the operation of the network computers. These projects include pipelined compiling[20], real-time control of complex processes, discrete[21] and continuous[20] simulations, demon language, and a FORTRAN compiler[22].

A pipelined compiler which produces distributed object code is described in this section to illustrate the network mode of operation and techniques to best utilize the network computer. A style of real-time control of complex processes appropriate to the network computer will also be presented.

### Pipelined Compilation and Problem Decomposition

To achieve efficient use of the network computer, the following strategies should be attempted:

a) A program must be partitioned into a number of parallel tasks.

b) There must be at least as many tasks as constituent computers.

c) The tasks may form clusters that have their own data structures, but between such clusters tasks must communicate only by sending and receiving messages since there is no common memory among computers.

d) The memory requirements of each cluster must not exceed the memory capacity of the constituent computer since memory management may be inefficient and difficult.

e) The amount of data that must be exchanged between clusters should be minimized to avoid saturating the communication channels.

f) The processing requirements among computers should be balanced and overlapped as much as possible, to minimize the overall execution time of the program.

These strategies can be illustrated by the compile-time and runtime environments of a DYNAMO[23] compiler[20] on the TECHNEC. Each compilation phase resides on a separate computer of TECHNEC. A source program is first entered from a console to a file at the system node. Statements of the source program pass through the phases in order, with no feedback required, so we may consider the compiler to be pipelined in the same sense as pipelined arithmetic units. The generated object code is decomposed into clusters by a partitioning program and stored in a file at the system node. When program execution is initiated, the clusters are loaded into the assigned processors. The program runs in a distributed mode.

Problem decomposition at compile-time and runtime is achieved by two approaches: functional partitioning and partitioning by data dependency.

The first approach divides a program into manageable portions according to its functions with minimum message communication and maximum use of the available parallelism. The compilation process is organized as a pipeline. Three major constraints are thus imposed on the design. First, to keep the pipeline busy, each phase processes one statement at a time. Once the statement is processed, the statement in its new converted form is passed to the next phase. Neither the input form nor the converted form will be available to the phase for future processing. Second, since the nodes of TECHNEC do not share memory, individual phases cannot access global tables. Information derived by each phase should be imbedded in the internal code which is routed to succeeding phases. Third, although the total memory of the network is 144K words, each phase can access only its 12K words of local memory.

The second approach partitions a program into parallel clusters according to data dependencies between tasks of the program, but the functional properties of the program are not taken into consideration. Execution of clusters in parallel reduces execution time, but message passing and a potential wait are necessitated when two dependent tasks are assigned to different clusters. For example, a compile-time module of the DYNAMO compiler partitions the object code into clusters so that the overall execution time of the program is minimum.

A brief introduction to DYNAMO will be given before the pipelined compiler is described. DYNAMO[23], as a continuous simulation language, models a system as a set of variables called LEVELs and their rates of change called RATEs. The changes of LEVELs and RATEs with respect to time are expressed as a set of difference equations. Another class of variables, AUXILIARYs, are used to help specify complex relationships between LEVEL and RATE variables. A fourth class of variables, SUPPLEMENTARYs, are used solely in output statements.

DYNAMO is nonsequential--i.e., statements can be written in any order without affecting the outcome of the program. The general pattern of execution of one cycle of simulation is first LEVELs, then AUXILIARYs and RATEs. At certain specified time points, SUPPLEMENTARY variables are calculated. LEVEL statements are independent of one another and can be executed in any order: and the same is true of RATE equations when they are later executed. AUXILIARY variables, however, are usually interdependent.

The compiler consists of eight phases, as shown in Fig. 4. The first four stages are quite conventional. The Scanner converts input text into internal tokens and the Macro Expansion Module expands all macro calls to system-or user-defined macros into internal token strings. The input text, together with errors discovered by the Scanner and Macro Expansion Module is passed to a file for listing. The statement in token form is then passed to the next phase, Symbol Table Manipulation, as a message. The Symbol Table Manipulation Routine converts each identifier token into a unique index into the Symbol Table. Since statements are nonsequential, an identifier may be used in the right hand side (RHS) of a statement before it appears on the left hand side (LHS) of a statement. A data structure showing the dependency among statements is necessary to recognize undefined identifiers and doubly defined identifiers. The parser converts each internal token string into Polish suffix form using a transition matrix.

Since DYNAMO is non-sequential, the Sequencing Module determines the order of execution of statements and initialization equations of some identifiers. This Sequencing order is determined by building successor lists, one for each identifier. Thus, an identifier on the LHS of a statement is a successor of each of the identifiers on the RHS, since the values of the RHS identifiers must be known before the LHS identifier can be calculated. When all the statements are encountered, a topological sort program is run. The output of the topological sort program is the sequencing order.

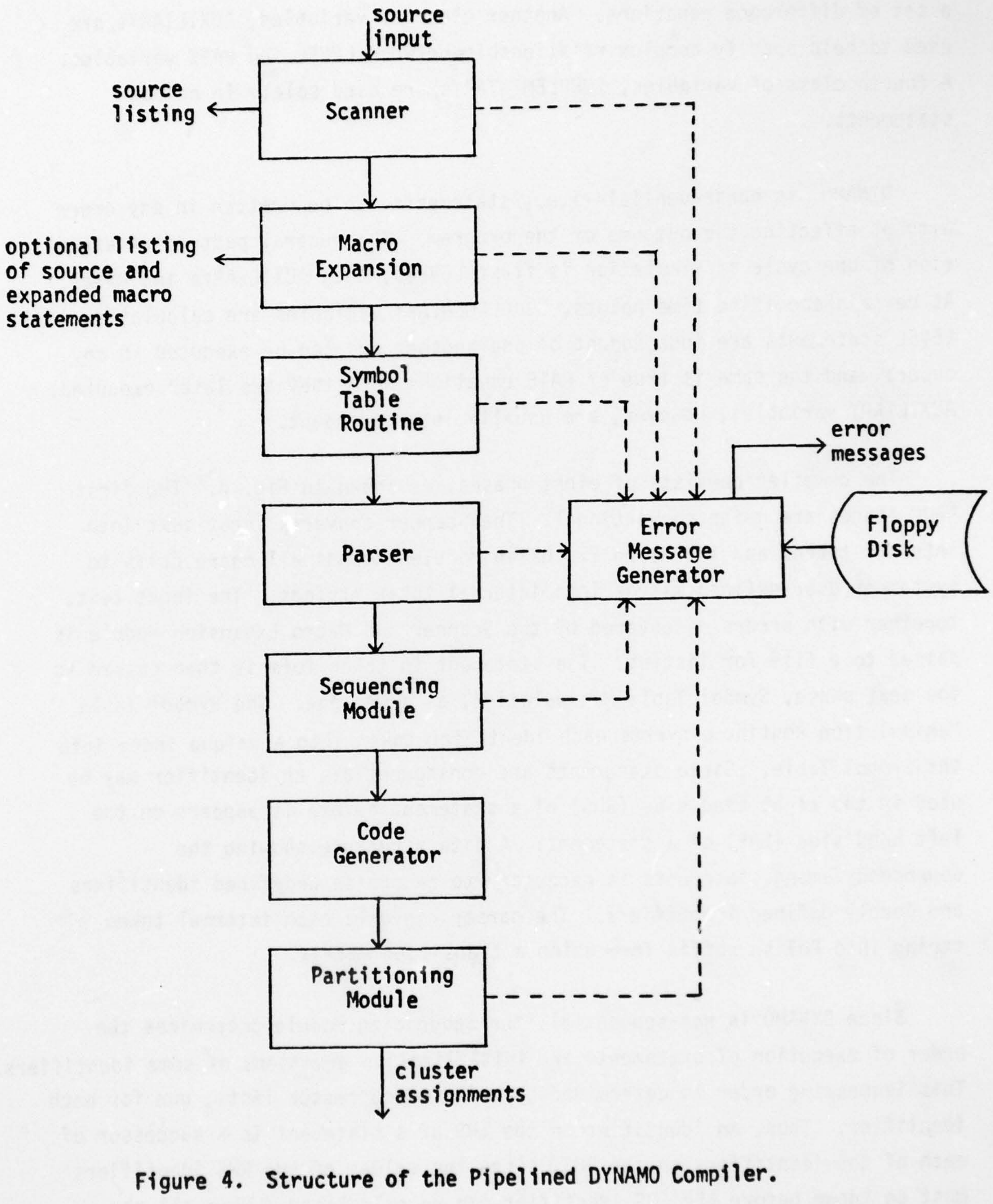


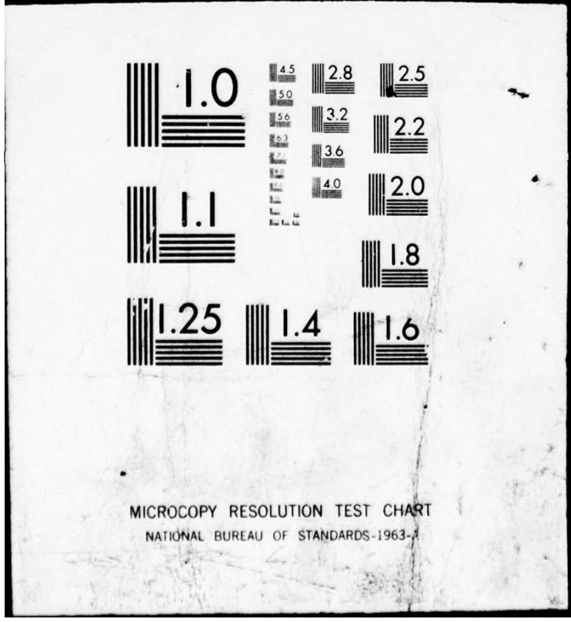
Figure 4. Structure of the Pipelined DYNAMO Compiler.

AD-A049 559 COLORADO STATE UNIV FORT COLLINS DEPT OF ELECTRICAL --ETC F/G 9/2
PROCEEDINGS OF THE ANNUAL ROCKY MOUNTAIN SYMPOSIUM ON MICROCOMP--ETC(U)
SEP 77 M ANDREWS, S MCCORMICK AFOSR-77-3367

UNCLASSIFIED AFOSR-TR-78-0017 NL

3 OF 4
AD
A049 559

049 55



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

The Sequencing Module can determine the sequencing order only after all statements are encountered. But both its predecessor, the Parser, and its successor, the Code Generator, function on a statement-by-statement basis. To utilize the full capacity of the pipeline, the Sequencing Module passes the Polish Suffix String to the Code Generator after extracting dependency information from it. The Code Generator transforms the Polish suffix string to a closed subroutine. The sequencing order, when supplied by the Sequencing Module, will be coded as subroutine calls.

In addition to generating code, the Code Generator reports memory requirement and execution time of each statement to the Partitioning Module.

The Partitioning Module aims to divide the object code into clusters of subroutines together with the associated calls. A mixed integer linear programming model produces optimal partitioning for all program structures.[24] The model assumes that every task (statement) is assigned to one and only one processor; and that communication time, if any, and execution time for each statement are known. No cluster exceeds the available main memory in size. The objective is to obtain a partition that will produce total minimum execution time for each simulation cycle. Other heuristics[24], based on the data dependency partitioning approach, are being tested against this optimal, but expensive, model.

One heuristic is to first assign a LEVEL quantity to a cluster of its own and gather its predecessors to the same cluster. If a quantity Q is the predecessor of more than one LEVEL, there are the options of duplicating Q and all predecessors of Q for each LEVEL, or assigning Q to one of the LEVELs and providing communication primitives to send the value of Q to others. In the second option, the "strength" of the dependency determines the assignment. The output of this phase will be statement subroutines, their calls, a list of processor assignments and appropriate communication commands. The latter two items of information are used by the loader for allocation of object code to processors.

The Symbol Table, the Sequencing Module, and the Partitioning Module all build and analyze data structures to determine dependency among statements.

Instead of a single complex data structure, each phase builds a separate local data structure. A centrally located data structure requires considerable accessing time while separate local data structures are more efficient and simpler because they are tailored for the specific needs of individual modules. Moreover, all three data structures are constructed and analyzed simultaneously.

At compile time, a statement in its internal form can be transmitted from one phase to another by point-to-point messages. The broadcast mode is convenient for disabling succeeding phases when a fatal error is detected.

At runtime, clusters on all processors are synchronized to start executing the same simulation cycle. The cycle synchronization signal can be generated by a signaling process which emits a broadcast message to poll each cluster's readiness to start a new cycle. Message transfers between clusters within the same cycle are carried out asynchronously by point-to-point messages. Readiness to start a new cycle implies completion of all executions and arrivals of all necessary values, if any, from other clusters for the next cycle.

## Complex Process Control

A major purpose of the TECHNEC Project is to use our network computer in developing a coherent style for controlling large-scale electronic, mechanical, or chemical systems in which so many variables must be regulated that real-time computation becomes difficult or impossible for a conventional computer of affordable size and speed. In general, we seek ways of delegating responsibility for control decisions to processors in a heterarchical control network, so that a program currently assuming an executive role will not be overwhelmed by having to control everything at once.

Skilled engineers use a variety of tricks and know-how to make such multivariable processes work. Well known examples[16,17] show many ways in which complex tasks requiring accurate coordination of many variables are best performed by distributed controllers, each handling a stage of rough computation, to avoid overwhelming an executive with the need to regulate a large number of variables at once. Underlying a large collection of these seemingly ad hoc engineering tricks for making complicated biological and artificial systems work may be discerned[16], a coherent style for controlling large-scale systems, in which authority and computations are delegated to loosely coupled subsystems that can fake what the system needs, each, perhaps, in a limited set of circumstances. The control problem consists in coordinating all these partial solutions into a coherent implementation of the solution in a wide variety of circumstances. Variety of response is achieved by adjusting, or "tuning" the partial solutions. In this way, the command for action can be a simple one that does not have to specify the precise variant of the action to be performed.

The partial solutions can be patterns of feedforward that maintain subsystems close enough to their desired configurations, so that very simple feedback mechanisms can achieve the small remaining corrections. This information-handling style reduces the staggering load on a controller that would result if it had to regulate all the degrees of freedom that participate in any complex action. Through such simple and imprecise stages of preprocessing, enough of the generality is removed from the world seen by a regulatory system, so that a regulator can succeed that is much too simple to succeed in a general world.[16] The specific problems of implementing these obvious ideas form the starting point for our theoretical study of heterarchical task organization,[18] which is a necessary part of creating an adequate high-level language of control actions (as distinguished from the variabilities of individual low-level realizations of these actions).

This style particularly suits computers that, without excessive computation, must regulate actions whose individual realizations must be infinitely variable (as parameters and conditions change), under the constraints of limited information and communication within a computer network, or within a large-scale integrated circuit chip whose components are not easily accessible from the outside. TECHNEC will enable us to experiment with this style of control, through parallel execution and control of the partial solution by the processers in the network, and to develop software (compilers, languages, simulation methods) for implementing this style.

As an example of distributed control of a multivariable system, we are simulating a robot arm. We program with a mind to applicability of our methods to the general kinds of control tasks that would be found in process control, reactors, spacecraft, etc. The arm is propelled by torques applied to its joints. These impulses produce ballistic ("freehand") movements, in which a movement, once started, is continued by the momentum of the arm, rather than the type of movement, common in numerical control and robot research, in which the arm is continuously guided through a succession of closely spaced points. Thus, we seek to make the arm follow a desired path by subjecting it to a small number of simply specified inputs (generated by small subroutines or simple components), rather than a large number of small impulses, each requiring a separate specification. Our main problem is to get from a description of the desired movement to a description of the necessary inputs.

The trend these days in practical and experimental robot arms is to use an arm having the minimum number of degrees of freedom (6) needed to place the hand in an arbitrary orientation at an arbitrary location. The reason is a desire to minimize computational complexity. However, this means that, to a given hand position there corresponds a unique configuration of all the joints, so that a desired path in space calls for a trajectory in joint-configuration space that (1) must be elaborately computed, and (2) is almost surely unlike any trajectory achievable through free ballistic movement, so that point-by-point control must be imposed.

In contrast, a human arm has many more degrees of freedom than the minimum, but when we swing our arm to pick something up, we lock most of these degrees of freedom, and use a cleverly chosen small subset of joint movements to produce an approximate realization of the desired path that can be achieved ballistically by a very small number of impulses. In this way, the existence of so many degrees of freedom does not complicate control in our style; rather, it allows the selection of small subsets of these degrees of freedom for use in simply conceived and executed recipes of movement. Thus, what you see as one arm, conceptually functions as perhaps 100 different "virtual arms". Each of these virtual arms has simple ways of doing particular kinds of movements. Our control system should maintain a catalogue of movement descriptions with pointers to their appropriate virtual arms and recipes for using them.

We are currently working on such a catalogue for handball-like task, subject to constraints of limited-precision observation and computation and the need to operate in real time. As a prerequisite, we have implemented an algorithm for simulating such an arm. Parallel processes in our network will embody the catalogue (with storage and retrieval programs), virtual arm "assembly", torque impulse generation, monitoring, and "demons" to notice significant conditions requiring interrupts or activation of subroutines.

## VI. Summary

We have attempted to explain the decisions made in arriving at the design of the network computer being constructed at Illinois Institute of Technology. Design alternatives and relation to other interconnected computer systems composed of minicomputers and/or microcomputers are examined. The network architecture and software facilities are presented briefly with emphasis on the message communication mechanism. Important strategies to utilize such a network computer are enumerated and illustrated by two applications. We believe that such a system opens new areas of application on a collection of loosely coupled microcomputers.

References:

1. Fuller, S. H. and D. P. Siewiorek, 'Some Observations on Semiconductor Technology and the Architecture of Large Digital Modules', IEEE Computer, vol. 6, no. 10, October 1973, 17-21.

2. Withington, F. G., 'Beyond 1984: A Technology Forecast', Datamation, January 1975, 54-73.

3. Fuller, S., D. Siewiorek and R. Swan, 'Computer Modules - An Architecture for a Modular Multi-microprocessor', Proc. ACM National Conference, 1975, 129-133.

4. Rattner, J. R., 'Microprocessor Architecture - Where do we go from here?', Proc. COMPCON Spring 77, 223-224.

5. Newell, A. and G. Robertson, 'Some Issues in Programming Multi-mini-processors', Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1975

6. Heart F. E., S. M. Ornstein, W. R. Crowther and W. B. Barker, 'A New Mini Computer/Multiprocessor for the ARPA Netork', Proc. National Computer Conference, 1973, 529-537.

7. Farber, D. J., J. Feldman, F. R. Heinrich, M. D. Hopwood, K. C. Larson, D. C. Loomis and L. A. Row, 'The Distributed Computing System', Proc. COMPCON 73, pp. 31-34.

8. Farber, D. J. 'A Ring Network', Datamation, Feb. 1975. vol. 21, no. 2, 44-46.

9. Farber, D. J. 'Distributed Data Bases - An Exploration', Proc. of the 1975 Symposium on Computer Networks: Trends and Applications, 25-27.

10. Fraser, A. G. 'A Virtual Channel Network', Datamation vol. 21, no. 2 Feb. 1975, 51-56

11. Wulf, W. A. and C. G. Bell, 'C.mmp - A Multi-Mini-Processor', Proc. FJCC 1972, vol. 41, AFIPS Press, 765-777.

12. Anderson, G. A. and E. D. Jensen, 'Computer Interconnection Structures: Taxonomy, Characteristics and Examples', Computing Surveys vol. 7, no. 4, Dec. 1975, 197-213.

13. Fuller, S. H., D. P. Siewiorek and R. J. Swan, 'Computer Modules: An Architecture for Large Digital Modules', Proc. of the First Annual Symposium on Computer Architecture. Dec. 1973. 231-237.

14. Farmer, W. D., and E. E. Newhall, 'An Experimental Distributed Switching System to Handle Bursty Computer Traffic,' Proc. ACM Symp. on Data Communications, Pine Mountain, Ga., October 1969, pp. 1-33.

15. Dertouzos, M. L., 'Control Robotics: The Procedural Control of Physical Processes', Information Processing 74 (IFIPS Congress 74), 1974, 804-813.

16. Greene, P. H. 'Strategies for Heterarchical Control--an essay. I. A Style of Controlling Complex Systems', Illinois Institute of Technology Technical Report, 1975.

17. Greene, P. H. 'Problems of Organization of Motor Systems'. In Rosen, R. and Snell, F. M. (eds.), Progress in Theoretical Biology, Vol. 2, New York: Academic Press, 1972. pp. 303-338.

18. Greene, P. H. 'Strategies for heterarchical control--an essay. II. Theoretical Exploration of Style of Control', Illinois Institute of Technology Technical Report, 1975.

19. Hazra, A., 'Simulating Open Chain Linkages: Implementing an Algorithm by Stepanenko and Vukobratovic' Technical Report 77-3, Department of Computer Science, Illinois Institute of Technology, 1977.

20. Huen, W., O. El-Dessouki, E. Huske, M. Evens, 'A Pipelined DYNAMO Compiler', Technical Report 77-5, Department of Computer Science, Illinois Institute of Technology, 1977.

21. Christopher, T., 'Transactions Oriented Queueing System Simulation on Network Computers', Ph.D. Dissertation, Department of Computer Science, Illinois Institute of Technology, August 1977.

22. Huen, W., P. Greene, R. Hochsprung, T. Christopher, F. El-Wailly, A. Hazra, K. Sista, 'TECHNEC, A Network Computer for Distributed Task Control', Technical Report 77-4, Department of Computer Science, Illinois Institute of Technology, 1977.

23. Pugh III, A. L., 'DYNAMO II User's Manual', MIT Press, 1975.

24. El-Dessouki, O. and W. Huen, 'Automatic Partitioning for a Network Computer', Technical Report 77-6, Department of computer Science, Institute of Technology, 1977.

# MACRO BASED HIGH-LEVEL LANGUAGE SOFTWARE FOR MICROCOMPUTERS*

by

Gearold R. Johnson
Associate Director for Research
University Computer Center
Associate Professor
Computer Science Department
Associate Professor
Mechanical Engineering Department
Colorado State University
Fort Collins, Colorado 80523

## ABSTRACT

The advent of the microprocessor once again raises the question of programming suitability. This paper deals with the development of semi-automatic generation of cross-compilers for writing high-level language cross-compilers to support microcomputers. The techniques used are the translation of the source language to a macro string for processing by a macro cross-assembler to produce machine dependent object code. The automatic production of cross-assemblers is discussed. This software should provide for rapid generation of cross-compilers which will result in greater usage of high-level languages for software development for microcomputers.

---

*This work was partially supported by the National Science Foundation Grant DCR75-03578.

## INTRODUCTION

The large scale integrated technology of the seventies has created a revolution in computing systems. First, there was the introduction of the microprocessor [1] in 1971. These early and primitive multiple chip CPUs were quickly followed in 1974 by the introduction of more sophisticated eight bit microprocessors such as the Intel 8080, Signetics 2650, Motorola M6800, etc. culminating in the current versions of these devices plus many new ones. In addition to the LSI impact on CPU elements, the same technology has produced the 16K dynamic ram and the 4K static ram resulting in 1977 memory prices of about $250 per 16K bytes. These prices indicate that current generation microprocessors which typically have direct addressing capability of 65K can be fully memory configured for approximately $1,000. In 1977, the CPU and memory chips were supplemented with the introduction of peripheral processors for use as disk controllers, keyboard controllers, etc. This will permit very low costs for data entry peripherals and data storage peripherals such as the keyboard-CRT terminal device and the floppy disk rotating mass storage device. LSI has therefore produced the $4,000 to $5,000 8-bit microcomputer with a relatively large amount of memory (65K) and with mass storage and input/output facilities. Such is the current state of the hardware.

Unfortunately, software development is lagging hardware development severely. Microcomputer software is following in the same footsteps as its minicomputer precedessor. First, the development of relatively crude resident assemblers followed by small operating systems (monitors) and then one or two relatively crude languages. This is illustrated in the microcomputer by the emphasis on such languages as BASIC [2].

One innovation in microcomputer software has been the use of larger machines as software development aids, that is, the large scale usage of cross-system software. Cross-system software [3] can be highly sophisticated because it takes advantage of the host machine's software capabilities. For example, cross-assemblers need not be crude but can include such features as found on maximachine assemblers, macro facilities, conditional assembly, large numbers of pseudo operations, etc. This allows full use of the maximachine to be used in a more effective manner for software development rather than using the crude resident assemblers. However, even many of the cross-system assemblers are little more than mnemonic translators.

Intel recognized the need for a systems language and their pioneering effort with the PL/M language is to be applauded. This language is used in-house for nearly all of Intel's software development. The first versions of PL/M were cross-compilers for the 8008 and 8080. PL/M has subsequently been imitated by other vendors to provide cross-compilers for their microprocessors. Examples are PLμS, MPL, and others.

If there are any characteristics that attach to the problems of programs run on microcomputers, it is that they are primarily systems programs. That is, they are programs with long life, they are well debugged, perhaps quite complex and often directly concerned with hardware. Moreover, microcomputers are often devoted to a single task. For example, as a standalone BASIC or APL system used in an educational environment, a data acquisition system, driving graphics displays or doing real-time laboratory analysis. In many instances, the task occupies nearly the full resources of the machine. However, there are many more where processing time is available but unused. The fact that most of these programs are systems oriented is evidenced by the software that has been developed for these machines. Therefore, there is certainly a need for high-level languages to support system software development.

But, the real problem is that software must be generated from scratch for each new microprocessor introduced into the market. The large number of microprocessors has made it paramount that new techniques be developed for the implementation of high-level languages for the numerous microprocessors that already exist and innumerable ones that will exist in the future. New software techniques are necessary. The GEN [4, 5] system developed at Colorado State University was an attempt to automate the production of microcomputer cross-assemblers and simulators. The same is required for high-level languages.

The compilation problem has been carefully analyzed and has been reduced to one now of requiring automatic code generation in order to automatically produce compilers; but, this remains an unsolved problem although there are a number of references [6-9] which describe the problem. It is the intent of the present author to pursue the development of high-level language cross-compilers using sophisticated cross-assemblers with macro and conditional assembly facilities in order to generate object code for different microprocessors.

Before proceeding to the discussion of the high-level language cross-compiler, the next section briefly introduces the ASM/GEN system for automatically producing cross-assemblers for microcomputers.

## DESCRIPTION OF ASM/GEN

ASM/GEN is a cross-assembler generating system for small computers [10]. The concept behind the program development was to construct a software system that would produce cross-assemblers from a user provided machine description. ASM/GEN consists of a single generator routine which generates the target assemblers with two modules of pre-written code (referred to as "skeleton decks") provided with the package and the user-provided input. The generated cross-assemblers are highly modular in their structure so as to make ad hoc modifications a more straight-forward process.

Other attempts to solve this problem have been made using a somewhat different approach. The approach was to develop a generalized (or universal) assembler called a meta-assembler [11-14). The meta-assembler is basically a machine independent cross-assembler which takes both the assembly source program and the machine dependent specifics as part of the input for each run.

The syntax of a meta-assembly program is characteristic of a typical assembly program. It includes a label field, op-code mnemonic field, operand mnemonic fields, and provision for typical pseudo directives. In addition, the programmer includes meta-directives stating the word size of the computer, number representation, and other machine specific descriptions. The machine instruction set syntax is described via a FORMAT directive. The FORMAT directive includes an identifier, the expressions to be assembled into object code, and the length and relative ordering of the expressions. The programmer references a given format by writing its identifier in the op-code (or command) field, and the expressions to be translated in the operand fields.

Another meta-directive is the PROCEDURE directive. The procedure has one or more identifiers and is referenced by writing any of them in the command field. Procedures can be used to generate one or more instructions and might be aptly described as a generalized macro facility.

Although both ASM/GEN and the meta-assembler solve essentially the same problem, the following differences should be noted:

1. ASM/GEN requires a description of the machine dependent data <u>only</u> in the creation run of the generated assembler, while the meta-assembler requires this information to be part of each assembly; and
2. ASM/GEN generates an assembler for a specific machine which then can be used repeatedly for the assembly translation process, while the meta-assembler itself accomplishes the assembly translation in a universal fashion permitted by the inclusion of the machine specific descriptions.

The most important consideration in ASM/GEN is the correctness and quality of the generated assemblers. The philosophy taken was to provide one standard set of assembly statement syntax conventions, constant specifications, symbol and macro table management schemes, and all pseudo instructions. There are several advantages and at least one disadvantage to this procedure. On the positive side, it most importantly minimizes the required user input and, thus, greatly reduces the assembler generating effort. Note that this is accomplished by consolidating all scanners, table management routines, pseudo instruction processors, etc. into the skeleton decks.

Another important advantage is in the standardization effect. Typically, microcomputer systems programmers may be constantly working with a variety of microcomputers due to rapidly changing technologies and the phenomenal diversity the market offers. Hence, if the ASM/GEN system is used to generate all assemblers, they will have a great deal in common. Those who have had to work in several computer languages concurrently can appreciate the advantage offered here by the standardized syntax and pseudo instructions.

One disadvantage of this scheme lies in the inability of ASM/GEN assemblers to process vendor specified assembly code. This lack of portability can be inconvenient and should be understood by those using the ASM/GEN system to avoid misguided and unnecessary expenditures of human energy! The point here is that the ASM/GEN system can build a well facilitated cross-assembler for a given microprocessor but <u>not</u> the vendor's assembler.

In order to generate a cross-assembler for any target microprocessor, the user provides to ASM/GEN as input the opcode and operand mnemonics, their binary translations, and the bit position information for a specific machine. This is used with the invariant sections of the assembler to generate a complete ANSI FORTRAN IV macro assembler in four distinct phases which are discussed below.

In the first phase, all invariant code (a collection of subroutines and functions) are transferred to the file to contain the complete assembler. This code accounts for about eighty percent of the entire assembler. The variant code consists of the routines which process assembly translation of the machine instructions on a class-wise basis (as described in [10]), a routine which directs the assembler to the proper translation class module for processing, and the complete data definition of all symbol table entries.

Phase one is accomplished by using a skeleton deck as standard ASM/GEN input. The skeleton deck includes all invariant program sections and a marker to flag the necessity for ASM/GEN intervention. A dollar sign in column one was used as the marker since it is an invalid FORTRAN usage. The phase one action can be described as

```
read card image (from skeleton file [SKEL1]
repeat while image [SKEL1] ≠ $
begin write card image (to assembler file);
      read card image (from skeleton file [SKEL1];
end
```

using an Algol-like language. In addition, the first user input card is read and stored. This card contains the number of translation classes and the word size of the machine in bits.

Phase two is the translation class routine builder. It reads the invariant statements of a class routine from skeleton file [SKEL2] and uses the same marker scheme used in phase one. The user input for each translation class consists of information characteristic of all instructions in a given class, followed by the mnemonics and hexadecimal equivalents for each machine instruction in the class. The characteristic information includes the class identification number, the maximum number of op-code and operand mnemonic fields allowable, the number of object code words generated per instruction, the internal bit position where the binary equivalent of each field is to be placed, and the size (in bits) of each field in the object code word.

The internal bit position of each field is specified via the imaginary word formed by concatenating all object words together in a left-to-right fashion. For example, if there are x bits per word and y words of object code generated for all instructions in the given class, then there are z = x*y bits in the imaginary word. The user specifies the right-most bit position of each field as it would be in the imaginary word. Bit z-1 is assumed to be the left-most bit of the first object word, and bit 0 is assumed to be the right-most bit of the last object word.

The last card of each translation class input group is END. This signals ASM/GEN to rewind the skeleton [SKEL2] file and begin the process over for the next translation class to be defined. A special case is the specification of the general mnemonics used in the operand fields or in expressions which are not to be confused with op-code mnemonics. These are designated by using class identification number 0, followed by each mnemonic and its binary equivalent. As with all other translation class input groups, it is terminated by an END card. The entire input deck is terminated by detection of an end of file card immediately following an END card.

Phase three begins after all user input has been read and the symbol table entries for all mnemonics and their binary equivalents constructed. The purpose of phase three is to construct the routine which maps a given op-code mnemonic into its class value and then branches to that translation class routine. Mapping is done via the symbol table and the branch is accomplished using a computer GO TO statement to a sequence of CALL statements, one for each translation class routine.

The final phase writes out all data definitions in the BLOCK DATA routine. The data includes the class mapping function table, all reserved symbol table entries, the machine specific constants, and all pointer initializations for the assembler. Thus, the completed macro assembler is left on a disk file for subsequent use.

The generated assembler operates on a two-pass basis to allow simple resolution of forward label referencing. The first pass reads the source statements, translates them, and writes out resulting information to an intermediate file. When a forward reference is detected, the entire source statement is written to the intermediate file. A code is used to distinguish between the two. Also, all error messages are recorded on an intermediate error file by using an error number and the statement number where it was detected. In addition, an error flag is set.

The second pass reads the intermediate file and transfers the information accumulated in the first pass to the listing and load block routines. Statements with forward references are translated and the resulting information is transferred to the listing and load block routines. Any errors detected in pass two are written to a second intermediate error file in the manner discussed above. When the end of the intermediate file is detected, the error flag is checked. If true, an error merge routine is invoked, merging and ordering error records by statement number. The error listing routine is called next and uses the error numbers as computed GO TO indices to print out intelligible diagnostics. Finally, the symbolic reference map is checked and, if true, the reference map is printed concluding the assembly.

The complete assembler contains most frequently needed facilities found in popular contemporary assemblers. They include the ability to reference the program counter value (PC), free formatting with colons delimiting labels, blanks or commas delimiting op-code and operand mnemonic fields, a semi-colon for operational end of statement delimiting, multiple labels on a single card, PC value assignment (*ORG), PC value increment (*DS), ASCII text string specification of arbitrary length (*TXT), label equivalence (*EQU), definition and assignment of special arithmetic type symbols (*DEFA, *SETA), use of +, -, *, / in arithmetic expressions, macro definition (*MACRO, *MEND), embedded and recursive macro generation, conditional assembly with the standard IF, THEN, ELSE control structure, ability to specify a constant using any radix r, $2 \leq r \leq 16$, a listing title designator (*TITLE), an extensive cross reference map (*CREF), and extensive error diagnostics.

ASM/GEN exists as both a batch-oriented system and an interactive version. The prerequisite knowledge required of ASM/GEN users is fairly minor. The assembler generating process is a straight-forward procedure which consists of two basic phases. They are: (1) the classification of machine instruction mnemonics, and (2) inputting the field specifications, the mnemonics and their corresponding hexidecimal values. Most vendor literature is organized in this manner so that converting the necessary input data to ASM/GEN format typically takes less than 30 minutes. Cross-assemblers with macro and conditional assembly features can be generated in less than one hour for any small machine.

Table I presents the necessary input information for the cross-assembler for the National SC/MP as an illustrative example.

```
SC/MP XASM
5 8
0
P0 0
P1 1
P2 2
P3 3
END
1 4 2
8 10 8 0
8 1 2 8
LD C0
ST C8
AND D0
OR D8
XOR E0
DAD E8
ADD F0
CAD F8
END
2 2 2
8 0
8 8
LDI C4
ANI D4
ORI DC
XRI E4
DAI EC
ADI F4
CAI FC
DLY 8F
END
3 3 2
8 8 0
8 2 8
JMP 90
JP 94
JZ 98
JNZ 9C
ILD A8
DLD B8
END
```

```
4 1 4
0
8
LDE 40
XAE 01
ANE 50
ORE 58
XRE 60
DAE 68
ADE 70
CAE 78
SIO 19
SR 1C
SRL 1D
RR 1E
RRL 1F
HALT 00
CCL 02
SCL 03
IEN 05
DINT 04
CSA 06
CAS 07
NOP 08
END
5 2 1
0 0
8 2
XPAL 30
XPAH 34
XPPC 3C
END
```

TABLE I

## MACRO-BASED HIGH-LEVEL LANGUAGES

Compiler technology, like the previously discussed assembler technology, also recognizes machine invariant and machine variant code. This has been pointed out by numerous authors. Lexical analysis, syntactical analysis, and language parsing are well understood and algorithms have been developed to represent machine independency. However, in the area of code generation and code optimization, machine dependencies play a major role. As stated earlier, automatic code generation would seem to be the answer to the compiler problem, particularly for microprocessors where it would be nice to have the same language features supported by a number of different commonly used machines. Bunza [ 6 ] has indicated a number of different schemes that could be used for code generation. These include macro expansion, hierarchical macro interpretation, pseudo machines and abstract models, interpretation of abstract machine code and special executive calls, code to code translation, and the operator data base and data classification scheme that he introduces himself. It would seem that, with the availability of cross-assemblers with full macro and conditional assembly features, the first four categories would offer an avenue of study for high-level language compilers. These are: macro expansion, hierarchical macro interpretation, pseudo or abstract machine models, and interpretation of abstract machine code.

At Colorado State University, three parallel programs are presently under investigation. All three of these approaches utilize the macro capability of ASM/GEN. The projects are increasingly complex. The first project uses the macro processor as a pseudo high-level language. It allows the programmer to program in macros rather than in machine language instruction. Berry and Johnson [15] developed a macro based language for programmable data acquisition systems. The language is called Data Reduction Language (DRL).

DRL was developed to essentially automate the process of invoking all the correct subroutines to perform a set of operations on variables and constants. This is accomplished primarily through conditional assembly and macro expansion.

DRL currently has a limited instruction set (six basic instructions) which include four arithmetic type instructions and two register modifiers. The six basic instructions are: LOAD, MOVE, ADD, SUB, MULT, DIVIDE. This limited vocabulary, however, is still quite powerful as will be illustrated with a typical user program. Table II gives a brief symbolic description of the consequence of each instruction.

| Name | Parameters | Action |
|---|---|---|
| LOAD | VARIABLE | AC←VARIABLE |
| MOVE | DESTINATION,SOURCE | DESTINATION←SOURCE |
| ADD | VARIABLE | AC←AC + VARIABLE |
| SUB | VARIABLE | AC←AC - VARIABLE |
| MULT | VARIABLE | AC←AC * VARIABLE |
| DIVIDE | VARIABLE | AC←AC / VARIABLE |

TABLE II: DRL INSTRUCTIONS

The variable described in Table II may be one of the two following:

1) Constant. A constant is defined at assembly time and occupies memory locations in a PROM.

2) System Parameter. This variable is essentially a channel number on the A/D converter. LOAD SYSPARM will invoke

a call to the digitizing routine at the appropriate channel number and then scale the input according to a precedence for that class of signal (i.e., temperature conversion from °K to °F, normalize incident energy into langleys, etc.).

A typical user data reduction phase might consist of measuring the collected energy from a solar collector by summing an incremental energy flux which has been absorbed in the working fluid from an $\dot{m}C_p\Delta T$ calculation. Let us further assume that a linear model of $C_p$ versus T has been constructed:

$$C_p = .7236 \text{ Btu/lb°F} + .0006188 \text{ BTU/lb°F}^2 \cdot T(°F)$$

This is valid for a 60/40 (water/ethelene glycol) mixture. The energy flux is then:

$$\dot{E}_c = \dot{m}C_p(T_{c_{out}} - T_{c_{in}}) \qquad \text{(per minute)}$$

Summing for some number of intervals k and letting the time interval be one minute yields:

$$E_c = \sum_{i=0}^{k} mC_p\ (T_{c_{out}} - T_{c_{in}}) \qquad (1)$$

To implement this reduction in DRL, one must first clear EPSUM, the energy partial sum. This may be done by defining a constant 0.0 in EPROM and moving this constant to EPSUM:

```
MOVE     EPSUM ZERO
```

The DRL to implement equation (1) is then:

```
LOAD     TCOUT            ; GET TEMP AT COLL OUTPUT
SUB      TCIN             ; SUBTRACT TEMP AT INPUT
MOVE     R1 AC            ; SAVE DELTA T IN R1
LOAD     TCOUT            ; GET COLL TEMP FOR CP CALC
MULT     .0006188         ; MULTIPLY BY SLOPE
ADD      .7236            ; ADD INTERCEPT
MULT     R1               ; MULTIPLY BY DELTA T
MULT     CMDOT          ; MULTIPLY BY COLL MASS FLOW RATE
ADD      EPSUM          ; ADD THIS FLUX TO TOTAL
MOVE     EPSUM AC       ; RESTORE ANS AS NEW EPSUM
```

The actual macro expansions of these DRL statements are shown on the next two pages. This language allows for rapid development of software for the programmable data acquisition system. This particular system is built around the MOS Technology 6502 microprocessor, but it could be rewritten for any microprocessor. Current plans are to extend the macro set to include a control structure so that the language can be used to write process control algorithms as well as data acquisition and processing functions.

The second project uses Halstead's [16] PILOT language which has been rewritten as a FORTRAN cross-compiler. The language has been slightly modified and redesignated the micro Pilot Language, μPL. The first pass over the μPL program converts the source code into a macro string list. This program consists of approximately 800 FORTRAN card images and took approximately two man weeks to implement. The compiler itself consists of two basic blocks: the lexical scanner and analyzer and the parser which uses a transition table parsing scheme based upon the current operator/next operator pair. Although the language is rather simple, the code for μPL is far more readable than the typical assembly language mnemonics as illustrated by the subroutine for a bubble sort shown on Figure 1.

The output macro string is made up from the following 17 macros: the program halt macro, STOPIT; the indexed addressing macro for handling subscripted variables, INDEXIT; the unconditional jump macro, BRANCH; the subroutine call macro, CALL SUBROUTINE; the load register macro, LOADIT; the output macro, WRITEIT; the input macro, READIT; the return from subroutine macro, RETURN; the arithmetic macros, ADDIT, SUBTRACTIT, MULTIT, and DIVIDEIT; the store register to memory macro, STOREIT; the equality comparison macro, EQUAL TO; the less-than comparison macro, LESS THAN; the data define and store macro, DEFINE STORE; and the open subroutine macro, SUBROUTINE.

A macro map of these 17 macros to target machine code completes the process. The macro and conditional assembly features of ASM/GEN are used for this pass. At the current time, the only operational program translates to 8080 machine code. The requirement here is that definitions for each of the required macros produced by the language have to be specified so that an individual can provide a macro map for any required microprocessor. This macro library is then attached to the ASM/GEN generated cross-assembler providing a high-level language for the specified microprocessor. Figure 2 illustrates the entire process.

Currently, the language is being extended to include boolean operators. The macro definitions are also being documented. In addition, the cross-compiler is being rewritten in μPL in order to self-compile the compiler. A simple resident macro assembler is also being written in μPL. This will eventually provide for resident μPL language capability on various microprocessors.

```
       !
       C0006:   40+16
                3
                61+16
                88+16
       !
       C7236:   0
                0
                72+16
                36+16
       !
       !SAMPLE OF A COLLECTED ENERGY ALGORITHM
       !
                LOAD     TCOUT          ! LOAD THE TEMP AT THE COLLECTOR OUTPUT
•••••           LDAI     TCOUT+20+16
•••••
•••••           JSR      DIGITIZ
•••••
•••••
•••••           JSR      TEMP
•••••
•••••
•••••           JSR      SWAP
•••••
                SUB      TCIN           ! SUBTRACT TEMP AT INPUT TO GET DIFF TEMP
•••••           LDAI     TCIN+20+16
•••••
•••••           JSR      DIGITIZ
•••••
•••••
•••••           JSR      TEMP
•••••
•••••
••••• .L3       JSR      FSUB
•••••
•••••
                MOVE     R1 AC          ! SAVE TEMPORARILY IN R1
•••••           LDXI     BYTENO+1
•••••
•••••           LDAAX    AC
•••••
•••••
•••••           STAZX    R1
•••••
•••••           DEX
•••••           BPL      F8+16
•••••
                LOAD     TCOUT          ! GET COLLECTOR OUTPUT TEMP AGAIN
•••••           LDAI     TCOUT+20+16
•••••
•••••           JSR      DIGITIZ
•••••
•••••
•••••           JSR      TEMP
•••••
•••••
•••••           JSR      SWAP
•••••
                MULT     C0006          ! MULTIPLY BY THE SLOPE
••••• .L4       LDXI     BYTENO+1
•••••
•••••           LDAAX    C0006
•••••
•••••
•••••           STAZX    TEMP1
•••••
•••••           DEX
•••••           BPL      F8+16
•••••
•••••           JSR      FMUL
•••••
```

```
•••••
                ADD     C7236           ! ADD IN THE INTEHSEPT
•••••   .L4     LDXI    BYTENO+1
•••••
•••••           LDAAX   C7236
•••••
•••••
•••••           STAZX   TEMP1
•••••
•••••           DEX
•••••           BPL     F8+16
•••••
•••••           JSR     FADD
•••••
•••••
                MULT    R1              ! MULTIPLY BY THE DIFF TEMP IN R1
•••••   .L4     LDXI    BYTENO+1
•••••
•••••           LDAAX   R1
•••••
•••••
•••••           STAZX   TEMP1
•••••
•••••           DEX
•••••           BPL     F8+16
•••••
•••••           JSR     FMUL
•••••
•••••
                MULT    CMDOT           ! MULTIPLY BY THE COLL. MASS FLOW RATE
•••••           LDAI    CMDOT+20+16
•••••
•••••           JSR     DIGITIZ
•••••
•••••
•••••           JSR     PRES
•••••
•••••
•••••   .L3     JSR  -  FMUL
•••••
•••••
                ADD     EPSUM           ! ADD TO THE ACCUMULATED ENERGY
•••••   .L4     LDXI    BYTENO+1
•••••
•••••           LDAAX   EPSUM
•••••
•••••
•••••           STAZX   TEMP1
•••••
•••••           DEX
•••••           BPL     F8+16
•••••
•••••           JSR     FADD
•••••
•••••
                MOVE    EPSUM AC        ! RESTORE THIS ANSWER INTO THE ACCUM ENER
•••••           LDXI    BYTENO+1
•••••
•••••           LDAAX   AC
•••••
•••••
•••••           STAZX   EPSUM
•••••
•••••           DEX
•••••           BPL     F8+16
•••••
```

FIGURE 1: BUBBLE SORT SUBROUTINE
WRITTEN IN μPL

```
SORT: {
        I <- -1,
        NTEST <- N-2,
    DO WHILE SWITCHED:
        SWITCHED <- 0,
    INNER LOOP:
        I <- I+1,
        K <- I+1,
        A[I] < A[K] ? INNER LOOP.;
        SWITCHED <- 1,
        TEMP <- A[I],
        A[I] <- A[K],
        A[K] <- TEMP,
        I < NTEST ? INNER LOOP.;
        SWITCHED =1 ? DO WHILE SWITCHED.;}
```

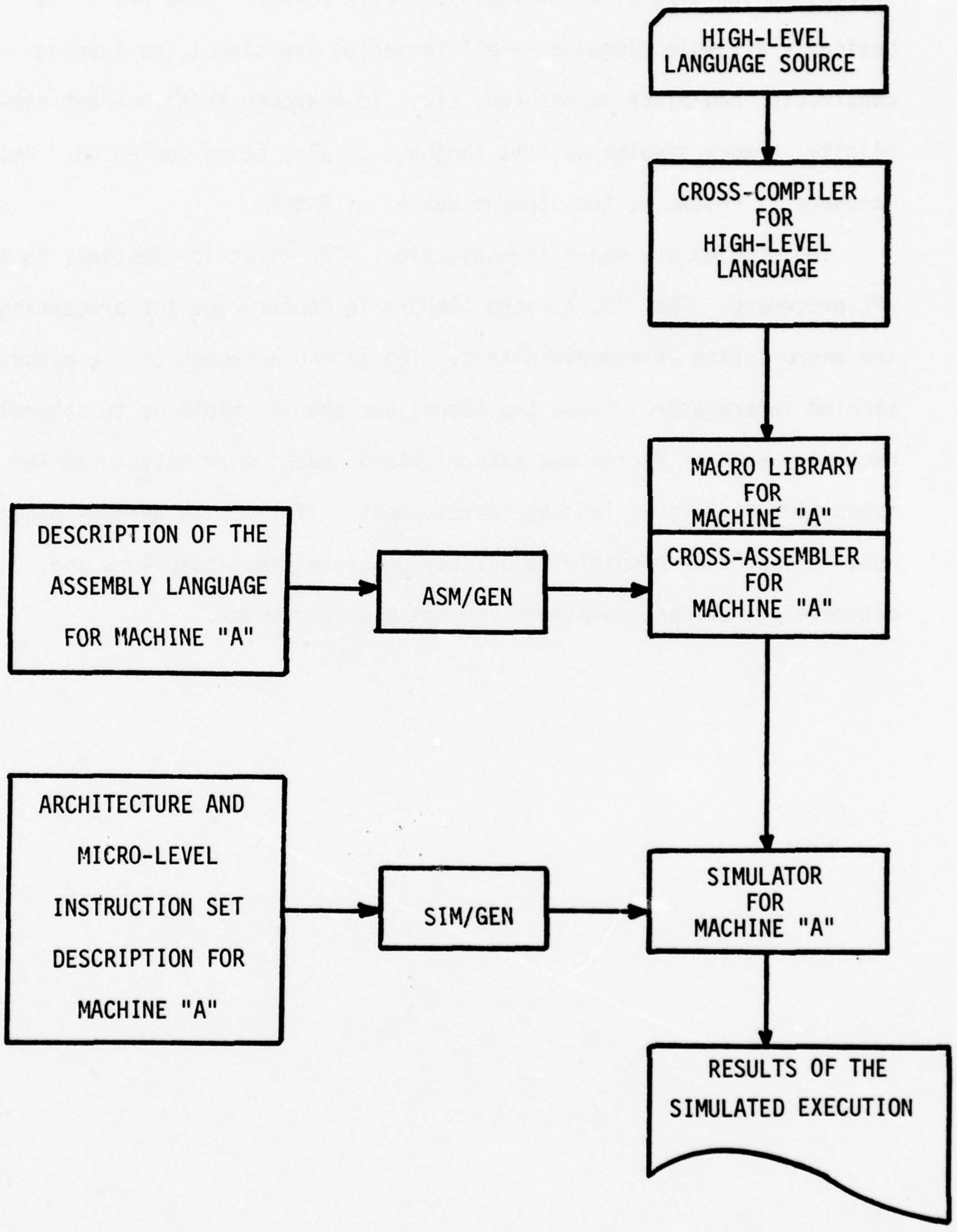

HIGH-LEVEL LANGUAGE SOURCE
CROSS-COMPILER FOR HIGH-LEVEL LANGUAGE
MACRO LIBRARY FOR MACHINE "A"
CROSS-ASSEMBLER FOR MACHINE "A"
DESCRIPTION OF THE ASSEMBLY LANGUAGE FOR MACHINE "A"
ASM/GEN
ARCHITECTURE AND MICRO-LEVEL INSTRUCTION SET DESCRIPTION FOR MACHINE "A"
SIM/GEN
SIMULATOR FOR MACHINE "A"
RESULTS OF THE SIMULATED EXECUTION

Figure 2: Use of GEN System Software

The μPL language is readily extendable in some ways by simply increasing the size of the transition table parser. However, it is basically a simple language -- all variables are global, no looping constructs, non-block structured, etc. To overcome this inherent simplicity, a more complex systems language is also being worked on. This language is PASCAL 5, the student subset of PASCAL.

Two studies are under investigation. The first is identical to the μPL procedure. That is, a macro library is constructed for processing the macro string intermediate text. The second approach uses a pseudo machine interpreter. These two techniques should enable us to determine necessary sets of macros and natural pseudo machine architectures for supporting high-level language development. If these techniques prove out, it should be possible to quickly generate cross-compilers and, ultimately, resident compilers for any microprocessor.

## REFERENCES

[1] Fagin, F., Shima, M., Hoff, M. E., Jr., Feeney, H. V., Jr., and Mazor, S., "The MCS-4 - An LSI Microcomputer System," Proceedings of the IEEE Region 6 Conference, 1972.

[2] Maples, M. D., Fisher, E. R., "BASIC for Intel's 8080," Micro 77 Computer Conference Record, Oklahoma City, Oklahoma, April 6-8, 1977.

[3] Watson, I. M., "Comparison of Commercially Available Software Tools for Microprocessor Programming," Microprocessors: Fundamentals and Applications. Edited by W. C. Lin, IEEE Press, 1977.

[4] Mueller, R. A. and Johnson, G. R., "A Generator for Microprocessor Assemblers and Simulators," Microprocessors: Fundamentals and Applications. Edited by W. C. Lin, IEEE Press, 1977.

[5] Johnson, G. R. and Mueller, R. A., "Automated Generation of Cross-System Software for Microcomputers," Computer, Vol. 10, No. 1, January, 1977.

[6] Bunza, G. L., "Automatic Multi-Target Code Generation for Microprocessors," Proceedings of the International Symposium on Mini and Micro Computers, Toronto, Canada, November 8-11, 1976.

[7] Miller, P. L., "Automatic Code Generation from a Machine Description," MAC TR-85, M.I.T. Project MAC, Cambridge, Massachusetts, May, 1971.

[8] Wilcox, T. R., "Generating Code for High-Level Programming Languages," TR71-103, Ph.D. Thesis, Cornell University, Ithaca, New York, NTISP13-203 582, September, 1971.

[9] Donegan, M. K., "An Approach to the Automatic Generation of Code Generators," Ph.D. Thesis, Rice University, 1973.

[10] Mueller, R. A. and Johnson, G. R., "ASM/GEN 5.1 Users Manual," University Computer Center Report UCC-76-2, Colorado State University, Fort Collins, Colorado, July, 1976.

[11] Ferguson, D. E., "Evaluation of the Meta-Assembler Program," CACM, Vol. 9, No. 3, March, 1968.

[12] Komincszk, C. P., "A Universal Cross-Assembler," M.S. Thesis, Report No. UIUCDCS-R-76-803, Department of Electrical Engineering, University of Illinois, 1976.

[13] Fizzarotti, E. and Berche, S., "Etude de Faisabilite d'un Meta-Assembleur," Direction des Etudes et Recherches - Service Informatique et Mathematics Appliques, Report HI/051-43012, April, 1970.

[14] Kasik, R. P., "The Design and Development of a General Cross Assembling Capability," Naval Underwater Systems Center, NUSC Technical Document 4994, Newport Laboratory, November, 1976.

[15] Berry, M. C. and Johnson, G. R., "The Design and Construction of a Solar House Data Acquisition System," University Computer Center Report UCC-77-2, Colorado State University, Fort Collins, Colorado, August, 1977.

[16] Halstead, M. H., A Laboratory Manual for Compiler and Operating System Implementation, Elsevier, New York, Americal Elsevier Publishing Company, Inc., 1974.

# RASTER SCAN CONVERSION USING CONCURRENT MICROPROCESSORS

Griffith Hamlin, Jr.
Institute for Computer Applications in Science and Engineering
Mail Stop 132C
Langley Research Center
Hampton, VA 23665

## ABSTRACT

Raster scan computer graphics displays require the image generated by the program to be converted to raster scan order. This is almost always done using a large frame buffer memory. An alternative technique is to substitute for the frame buffer memory enough processing power to perform the conversion "on the fly" for each frame. It appears that microprocessors can now provide this processing power at low cost. One possible implementation of such a raster-scan conversion algorithm is presented which uses one LSI microprocessor and one small special purpose processor running concurrently in a pipelined fashion. With today's microprocessor technology, this approach is shown to be feasible and its economics compare favorably with a frame buffer system of similar performance.

This report was prepared as a result of work performed under NASA Contract No. NAS1-14101 while the author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

## I. INTRODUCTION

Computer graphics display devices can be classified as either random or raster scan devices. Random scan devices allow the image to be drawn on the display in any order generated by an application program. For line drawings, this is often specified by a list of vector endpoints. Raster scan devices are constrained to display the image according to some specific order, usually left to right along horizontal scanlines. Therefore, before displaying an image generated by a program, the data must somehow be sorted so that it is available to the raster display device in the proper order. This raster scan conversion process poses an added complexity for raster scan displays. However, once this conversion is accomplished, raster scan displays have some advantages over random scan.

First, constraining the deflection of a CRT display to follow a fixed raster-scan pattern considerably simplifies the analog deflection electronics needed. Ordinary television receivers are raster scan CRT devices. Compared to available random scan computer display devices they are inexpensive, provide color, and require much less adjustment of the analog deflection circuitry. Also they are widely used.

Second, the time required to display one frame with raster scan devices is a constant (1/30 sec usually). Random scan devices, however, usually require a display time roughly proportional to the total length of all vectors being displayed. Complex pictures may require too much time to display (over 1/30 sec) and therefore appear to flicker. On raster scan devices an arbitrarily complex image can be displayed without flicker provided it is specified within the resolution limits of the raster scan display. Thus raster scan devices are usually used when it is desirable to display surfaces, which require considerably more displayed vector length than corresponding line drawings. However, raster scan devices suffer from a "stair stepping" effect when used to draw non-horizontal or non-vertical lines. Random scan devices do not suffer from this effect.

## II. APPROACHES TO RASTER-SCAN CONVERSION

A frame buffer memory is almost universally used to accomplish the raster-scan conversion process. buffer memory. One word of this memory is assigned to each resolvable (x,y) position on the display so that increasing addresses scan the display screen in the raster-scan order. The contents of any one word of this memory specify the intensity/color of the associated position on the screen. The frame buffer memory is loaded in any order required by the program with intensity/color information describing an image. The raster-scan output is then produced by scanning the memory sequentially from its lowest to highest address. The operation performed here is actually a "bucket sort" with each word of the frame buffer being one bucket capable of holding one datum.

Using a frame buffer memory with high resolution or many intensity/color levels requires much memory. For example, a 512x512 resolution with 512 intensity/color combinations requires 512x512x9 = 2,359,296 bits of memory. Also, to provide real time motion, each consecutive frame may display different images. For this, the memory speed must be fast enough to allow a new image to be written into the frame buffer within one frame time (1/30 sec). This writing time is a function of the complexity of the picture. In general, writing must proceed one word at a time since access is random. Of course, if real-time motion is not desired, the frame buffer can be filled slowly, after which it can be displayed for many frames.

An alternative approach is to perform the raster conversion process by sort techniques that do not require a large memory. With this approach, enough processing power is required for the entire sort to be done "on the fly" for each frame, even for systems without real-time motion. This approach therefore would seem to be useful for systems which need real time motion. Also, the complexity of a moving image that can be handled in real time is now determined by the speed of this processing rather than by the speed of the frame buffer memory. Using the approach described below, the speed of the processors must grow linearly with resolution to display an image of a given complexity. Frame buffer memory size and speed grow as the square of the resolution. Thus the processor sort approach appears useful for systems requiring high resolution. Jordan and Barrett[1] proposed one such conversion algorithm for line drawings. Earlier, Erdahl[2] described the design of hardware for executing the last portion of the scan conversion process for surface drawings. More recently, Meyer[3] has reported on a system in operation with hardware for this purpose.

This hardware, made by Staudhammer and associates[4], generates video in real time from run length encodings of the images.

Large frame buffers for high resolution diplays are becomming economically feasible due to the dropping cost of memory for frame buffers. However, processor cost is also dropping with the advent of inexpensive microprocessors. In the following section, a method of raster-scan conversion is presented which could be implemented using several small processors running concurrently. Such a system would be capable of moderate resolution and real time motion of moderately complex images. It is argued that the processors now becomming readily available have the capability of performing the raster scan conversion process and that because of their cost relative to memory costs, this approach currently compares favorably with a frame buffer system of similar parameters.

## III. RASTER-SCAN CONVERSION PROCEDURE

In order to show the feasibility of using concurrent processors to implement the approach described in this section, we will assume reasonable resolution and picture complexity parameters, determine the required processing speeds, and then present one possible design using these processors that could be implemented from readily available microprocessor components. Finally we make a comparison of hardware requirements of this implementation with the requirements of a frame buffer implementation. Specifically,

1. Assume 512x512 resolution. This is adequate for many purposes.

2. Assume 9 bits of intensity/color levels (say 8 levels of each of 3 colors).

3. Assume the picture complexity is at most 2000 straight vectors (for line drawings) or 2000 surface edges (in the case of surface drawings). This number was obtained by counting lines on several drawings of aircraft and spacecraft obtained from engineers involved in vehicle analysis at NASA Langley Research Center.

4. Assume the maximum number of vectors (surface edges) that intersect any horizontal scan line is 500. This is 25 percent of the entire picture. The drawings mentioned in (3) above had at most 13 percent of their vectors on any one scan line.

5. Assume a refresh rate of 30 frames/second.

In order to compare the cost of implementing this raster-scan conversion method with the frame-buffer method, we must be able to compare processors with some equivalent amount of memory. A quick search through the microcomputer literature at the time of this writing reveals that a fast microprogrammable processor in the class of the INTEL 3000 series[5] or AM 2900 series[6], can be obtained for approximately the cost of 24K bytes of MOS memory. A processor of this class with an appropriate word size will hereafter be called a "fast microprocessor". Such processors are today available on a few LSI circuits, are microprogrammable, can be implemented with any convenient word size (bit sliced), and can execute 5 to 10 million microinstructions per second. Their cost relative to memory cost may or may not remain constant in the future. At a low level both processors and memory bits may be regarded as some number of logic gates to be fabricated onto one LSI integrated circuit. For this reason one might expect the costs of processors and memory to be at least somewhat correlated.

## A. INPUT DATA AND Y-SORT PROCESSOR

The input data describe, in an encoded manner, the image to be converted to raster-scan. This consists mainly of the endpoints of vectors along with their intensity/color. To generate surface images, these vectors are taken to represent the left edge of the surface, in a manner described in section C. For this example, endpoints are given as pairs of 9 bit integers of (x,y) screen coordinates. A vector from (Xs,Ys) to (Xe,Ye) of intensity I is described by 5 9-bit words consisting of:

| | | | | |
|---|---|---|---|---|
| Xs | Ys | Xe | Ye | I |

Without loss of generality, assume Ys $\leq$ Ye. We assume the existence of some computer capable of generating this list every 1/30 second if real time motion is required of the entire image. All transformations (rotation, scaling, etc.) are assumed to have been performed on this data. Alphanumeric data and other graphics commands could easily be accomodated, but are not relevant for this discussion.

The raster-scan conversion procedure described here consists of several sort and merge operations on the image data. Referring to FIGURE 1, we first sort the input data into ascending order of Ys, using a Y-sort microprocessor. This produces the Y-sorted vector list. Next, using a scan line processor, we produce a standard raster scan video signal. Each of the pipelined processors communicates with the next one by shared memory buffers. Double buffers are used so that the Y-sort processor can be processing frame n+1 while the scan line processor is processing frame n from the second buffer. For the Y-sort processor, a bucket sort would be appropriate with 512 buckets, each of variable size. This suggests a data structure consisting of a set of 512 linked lists, one list corresponding to the Y value of each scan line. Also, while sorting, the Y-sort processor should replace Xe with dx/dy = (Xs-Xe)/(Ys-Ye), calculated to 18 bits percision. A moment's reflection will show that 18 bits are needed to specify the slope with the same precision contained in the original data. To process 2000 vectors in 1/30 second requires a processor fast enough to process one vector each 33 us, on the average. This corresponds to about 200 to 300 instruction executions. A count of executed instructions in a small program written for the INTEL 3000 series microprocessor shows that this "fast microprocessor" can handle the sort, slope calculation, and linked list manipulation in the required time. This microprogram performed the division in about 150 instructions, leaving 50 to 150 instructions for the rest of the processing. The memory requirements of this Y-sort are 24000 9 bit words. This provides for two buffers each capable of holding the sorted data lists. Double buffering is used so that the Y-sort processor can sort the data for frame n+1 using one buffer while the scan line processor (described below) processes data for frame n using the other buffer. Each list entry consists of five 9-bit words of data and one 9-bit link pointer to the next entry in the list. Ys need not be stored with each vector.

## B. SCANLINE PROCESSOR FOR LINE DRAWINGS

As each scan line is processed, an ACTIVE LIST of all vectors intersected by the current scan line is maintained. New entries to the active list are taken from the top of the sorted vector list produced by the Y-sort processor. Vectors are deleted from the active list while processing the last scan line which intersects them. Each entry in the ACTIVE LIST consists of:

| | | | |
|---|---|---|---|
| Xc | dx/dy | Ye | I |

where Xc is initially set to Xs. For line drawings using this method, there is no need to sort the ACTIVE LIST. Hence the scan line processor simply processes each entry in the previous scan line's active list and then processes entries at the top of the sorted input data list (if any) that are intersected by the current scan line. The processing done to an entry from either of these two sources is the same. It consists of:

1. Calculate $Xc' = dx/dy + Xc$.
2. Place intensity/color I in a 512 word scan line buffer at all x-locations between Xc and Xc'.
3. If this vector will be intersected by the next scan line (i.e. $Ye \neq$ y-value of current scan line) place this vector into a second ACTIVE LIST buffer, replacing Xc by Xc'. Otherwise drop this vector from the ACTIVE LIST by not placing it into the second buffer. This second buffer will be used as the primary buffer on the next scan line.

The memory requirements for this process consist of two buffers to double buffer the active list each consisting of 2500 9-bit words. Also, two 512 word scan line buffers for holding the intensities (the result of this process) are needed.

The speed requirements of this processor are rather high. For a maximum size ACTIVE LIST of 500 entries, the processor must process one entry approximately every 130ns. With today's "fast microprocessor" speeds, this is an impossible situation. For realistic images, this maximum size should seldom be reached, thus relaxing the speed requirement and allowing the use of one or more "fast microprocessors" if one is willing to use a statistically average length active list and several 512 word scan line buffers to feed the video generator while processing scan lines with long active lists. This would normally have no effect on real time motion of the images. However, if all line buffers were empty when the video generator requested the next line, the display could not continue at the normal rate. An unmodified TV display will usually not operate at a reduced rate. Some type of display with a variable scan line processing rate would be well suited for this approach.

However, a relatively simple special purpose hardwired processor can perform the required function for 500 vectors in real time for each line. A design of such a unit is given in FIGURE 2. For comparison purposes, we will assume two scan line buffers and the special-purpose hardware processor of FIGURE 2. A more detailed design of this processor has been done using the readily available 7400 series logic family. It is implementable for about the hardware cost of implementing a "fast microprocessor" CPU.

To actually drive a TV or other raster device, a hardware video generator will also be needed which will accept one 512 word buffer of intensity information for each scan line and generate the video signal. A similar video generator is needed for the frame-buffer method also, so its cost will not be considered in comparing the methods.

## C. SCANLINE AND ACTIVE-LIST PROCESSORS FOR SURFACES.

Shaded surface images can be processed in much the same manner as line drawings. In this case we assume that each vector in the ACTIVE LIST represents the left side of a planar polygon. This surface is assumed to extend to the right until reaching the next line to its right in the ACTIVE LIST. This requires the ACTIVE LIST to be sorted on Xc from left to right. Note that this representation of surfaces does not contain a separate right side for polygon boundaries, so overlapping surfaces cannot be represented. If the image data should contain two overlapping surfaces, one or the other of the surfaces must be displayed. If we do not display surfaces of less than one raster unit in width (below the display resolution), then the integer part of all Xc values for all vectors in the active list at any one time will be different (except for overlapping surfaces with a common left side). Thus the active list can be stored as a continuous vector in memory, using 512 consecutive addresses with each address associated with some integer value of Xc. New entries can be easily placed in the correct position in the active list. Entries may change places each scan line as Xc is updated.

The scan line processor for this scheme is similar to the processor for line drawings. A block diagram is given in FIGURE 3. This processor also merges new active list entries for scan line n+1 into the active list while processing scan line n. These new entries are easily placed directly into the proper position in the active list. With 2000 vectors spread over 512 scan lines there will only be about 4 additions per scan line on the average. However, this number could vary up to 500 additions for some unusual images.

A design that handles either lines or surfaces is only slightly more complex than either FIGURE 2 or 3, and would be the more reasonable implementation choice. It is simply the union of the main parts of both FIGURES 2 and 3 with a few switches located at points where the two diagrams differ.

Since the active list data is sorted on Xc there is no need for the 512 word scan line buffers as before. Instead a single word register contains the current beam intensity/color. The processing of the active list can be synchronized to the current X-value of the raster scan. As the raster sweeps across a scan line from left to right, X changes by one unit each 132ns. The current active list buffer is scanned in synchronism at this rate. When encountering a non-empty entry (i.e. the left side of some surface is encountered), data is loaded from the active list into a register. The I portion of the data in this register continually specifies the beam intensity/color. An adder (which easily works in 132ns) adds dx/dy to Xc, and the register contents are placed in a new active list buffer at location

Xc+ dx/dy (provided Ye ≠ current scan line y-value). This new active list buffer will be the input buffer for the next scan line.

The memory required for scan line processing of surfaces is two 2048 9-bit word active list buffers (we need not explicitly store the integer part of X for each active list entry). No 512 word line buffer is needed.

The cost (complexity) of such a special-purpose processor as estimated from a design using 7400 series logic, is approximately the same as for the processor described in section B. A comparable video generator is also needed as in B.

## D. POSSIBLE MODIFICATIONS/ENHANCEMENTS TO THE PROCESS

If the scan line processor was implemented in software or used a slower inexpensive microprocessor that was not able to execute the algorithm within one frame time on some complex pictures, a modified design could be used so that the X position on each scan line where I changes value would be stored in an encoded manner in a buffer. The length of this buffer is proportional to the picture complexity, and is generally much smaller than a frame buffer memory. It could be used to keep the display refreshed for many frames by a relatively simple hardware device to generate the video signal. This, of course, precludes displaying a different image on each frame.

One advantage of partitioning the processing between the Y-sort and the scan line processors in the manner described above is that several hidden surface algorihms which produce raster scan output[7,8,9] use, at an intermediate step, an ACTIVE LIST containing the data sorted in the same order as the ACTIVE LIST described above. Thus it would be possible to replace the scan line processor described above with a more complex processor that would discover the overlapping surfaces from the ACTIVE LIST and produce a display with hidden surfaces removed. The Y-sort processor could still be used to produce the active list. In this case the I value in the ACTIVE LIST would normally contain an identification number for each surface. Also, an even number of entries would appear in the ACTIVE LIST for each surface, corresponding to the edges where the scan ray enters and exits the surface as it moves left to right along one horizontal scan line. The intensity of each different surface is supplied by indexing in an intensity table, using the surface identification number as index. If such a processor was not capable of processing the entire image in one frame time, the intensity change buffer just described could be used to buffer the image for several frames.

## IV. COMPARISON OF FRAME-BUFFER AND PROCESSOR METHODS

Table 1 compares the performance limiting factors of the frame-buffer and processor approach. A frame buffer for 512x512 resolution requires 262,144 9-bit words of memory. The processor approach described in this paper requires only 30,000 9-bit words of memory (29,000 words for surfaces), or about 1/9 as much memory as the frame buffer method.

The frame buffer approach requires some processing power to generate the intensity patterns for vectors from their endpoint descriptions. To meet the 2000 vector per frame specification this requires a processor capable of processing 1 vector and storing the results in a frame buffer each 17us. In the processor approach, the scanline processor performs this function on the fly. The main difference between methods here is that the processor approach must do this calculation in real time, whereas the frame buffer approach may do it more slowly at the expense of real time motion.

The frame buffer approach has no component corresponding to the Y-sort processor. Therefore, the equipment tradeoff between the two methods is a Y-sort "fast microprocessor" and a scan line hardware processor vs. 232,000 9-bit words of memory and enough host processing power to generate the intensity patterns for vectors. Since the scan line processor performs essentially the same algorithm , we may, to a first approximation, equate the scan line processor to the cost of the host processing power needed to generate the intensity frame buffer patterns for individual vectors. An informal survey of the current literature shows that 232,000 words of memory has a cost many times that of a "fast microprocessor". We are considering only component costs here, supposing the fabrication costs for 232,000 words of memory is approximately equal to assembly costs of a "fast microprocessor". This assumption is based on today's approximately equal integrated circuit count for both the memory and processor described by FIGURE 3.

## V. SUMMARY

The algorithm and suggested implementation using microprocessors is not proported to be the best such algorithm or implementation. However, it does show the capability of a microprocessor and a small special purpose processor to perform the raster scan conversion process. Thus the use of this technique appears feasible. Economically, we conclude that unless the ratio of processor to memory costs changes drastically from its current value, implementation without a frame buffer appears to be preferred, based on today's component costs, for systems wih high resolution or real time motion. Less readily comparable differences in the two raster conversion processes are the maximum picture complexity limits imposed by processor speeds vs. the maximum real-time motion picture complexity imposed by frame buffer memory speed and host bit-map generating speed. Also not readily comparable are the differences in software required by the loss of a frame buffer and the addition of a vector list describing the image. The frame buffer allows easy reading of the current image at a given (x,y) point. Lieberman[10] notes that this makes it easy to discover the edges of any enclosed region in the image, or to find one's way out of a maze. On the other hand, the existence of a vector list describing an image allows transformations to be performed easier. A frame buffer system by Garrett[11] even includes a vector list for this purpose. Note that in the approach described in this paper, real-time motion of the entire displayable image is automatic, provided the host computer or yet another dedicated microprocessor can generate the input data in real time. On the other hand, with a frame buffer, any arbitrary memory intensity/color pattern can be displayed (flicker free), even with moderately slow memory, so long as it does not all move in real time.

| Performance parameter effected. | Limited in frame-buffer approach by: | Limited in processor approach by: |
|---|---|---|
| XY resolution | size of frame buffer | speed of scanline processor. |
| Complexity of still picture | No limit within resolution. | Speed of all processors |
| Complexity of real-time motion images. | Ability of host to generate coordinates in real time. Speed of host CPU to interpolate between vector endpoints. | Abiliy of host to generate coordinates in real time. Speed of scanline processor. |
|  | Write speed of frame buffer memory to accept new image bit map. | Speed of Y-sort processor. |

TABLE 1.
FRAME BUFFER - PROCESSOR APPROACH COMPARISON

## REFERENCES

[1] Jordan, B.W., and R.C. Barrett, "A Scan Conversion Algorithm with Reduced Storage Requirements", Communications of the ACM 16,11 (November 1973), pp. 676-681.

[2] Erdahl, Alan C., "Displaying Computer Generated Half-Tone Pictures in Real Time", Computer Science Department, University of Utah, RADC-TR-69-250, 1972.

[3] Myers, A.J., "A Digital Video Information Storage and Retrieval System", Third Annual Conference on Computer Graphics, Interactive Techniques, and Image Processing, Philadelphia, Pennsylvania, July, 1976, pp. 45-50.

[4] DIGITEK, Inc., Box 5486, Raleigh, North Carolina, 27607.

[5] Schottky Bipolar LSI Microcomputer Set, INTEL Corp, 3065 Bowers Ave., Santa Clara, California.

[6] AM2901, AM2909 Technical Data, American Micro Devices, Inc., 901 Thompson Place, Sunnyvale, California 94086.

[7] Bouknight, W.J., "A Procedure for Generation of Three-Dimensional Half-Toned Computer Graphics Presentations", Communications of the ACM, (13,9), September, 1970, pp. 527-536.

[8] Romney, B.W., G.S. Watking, and D.C. Evans, "Real-Time Disply of Computer Generated Half-Tone Perspective Pictures", Proceedings of the IFIP Congress 1968, pp. 973-978.

[9] Hamlin, G.A., C.W. Gear, "Raster-Scan Hidden Surface Algorithm Techniques", Proceedings of the Fourth Annual Conference on Computer Graphics, Interactive Techniques, and Image Processing, San Jose, California, July 1977.

[10] Lieberman, Henry, "The TV Turtle: A Logo Graphics System for Raster Displays", Proc. ACM Symposium on Graphics Languages, Miami Beach, Florida, April 1976, pp. 66-72.

[11] Garrett, Michael T., "An Interpretive/Interactive Subroutine System for Raster Graphics", Proc. ACM Symposion on Graphics Languages, Miami Beach, Florida, April 1976, pp. 101-105.

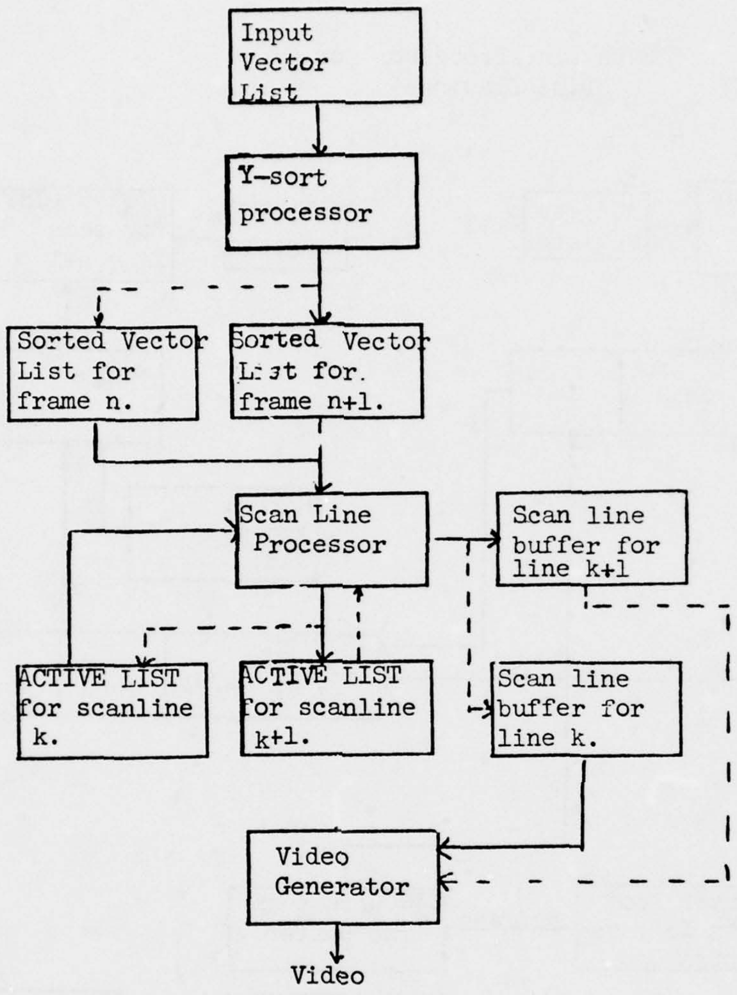

Input Vector List
Y-sort processor
Sorted Vector List for frame n.
Sorted Vector List for frame n+1.
Scan Line Processor
Scan line buffer for line k+1
ACTIVE LIST for scanline k.
ACTIVE LIST for scanline k+1.
Scan line buffer for line k.
Video Generator
Video

FIGURE 1

Scan Line Processor for
Line Drawings

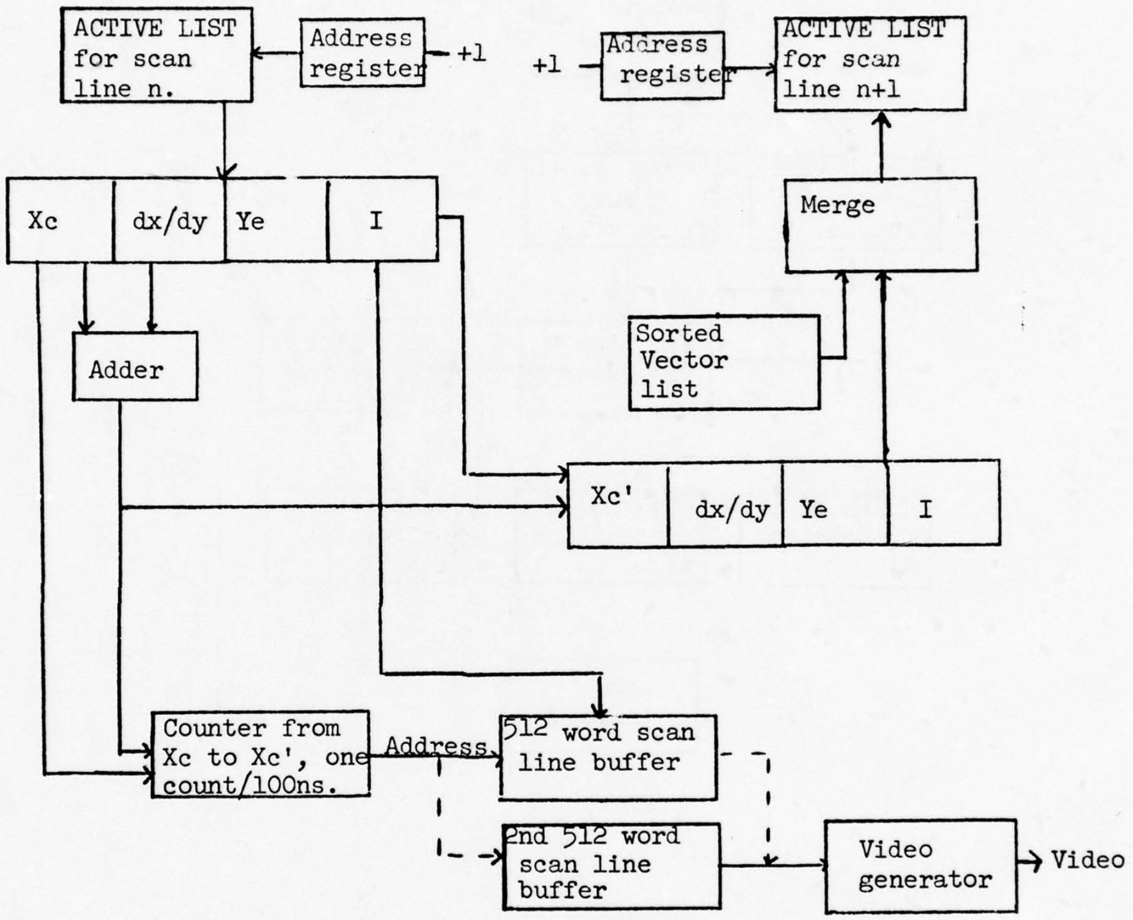

ACTIVE LIST for scan line n.
Address register
+1
+1
Address register
ACTIVE LIST for scan line n+1
Xc
dx/dy
Ye
I
Merge
Adder
Sorted Vector list
Xc'
dx/dy
Ye
I
Counter from Xc to Xc', one count/100ns.
Address
512 word scan line buffer
2nd 512 word scan line buffer
Video generator
Video

FIGURE 2

Scan Line Processor for
Surface Drawings

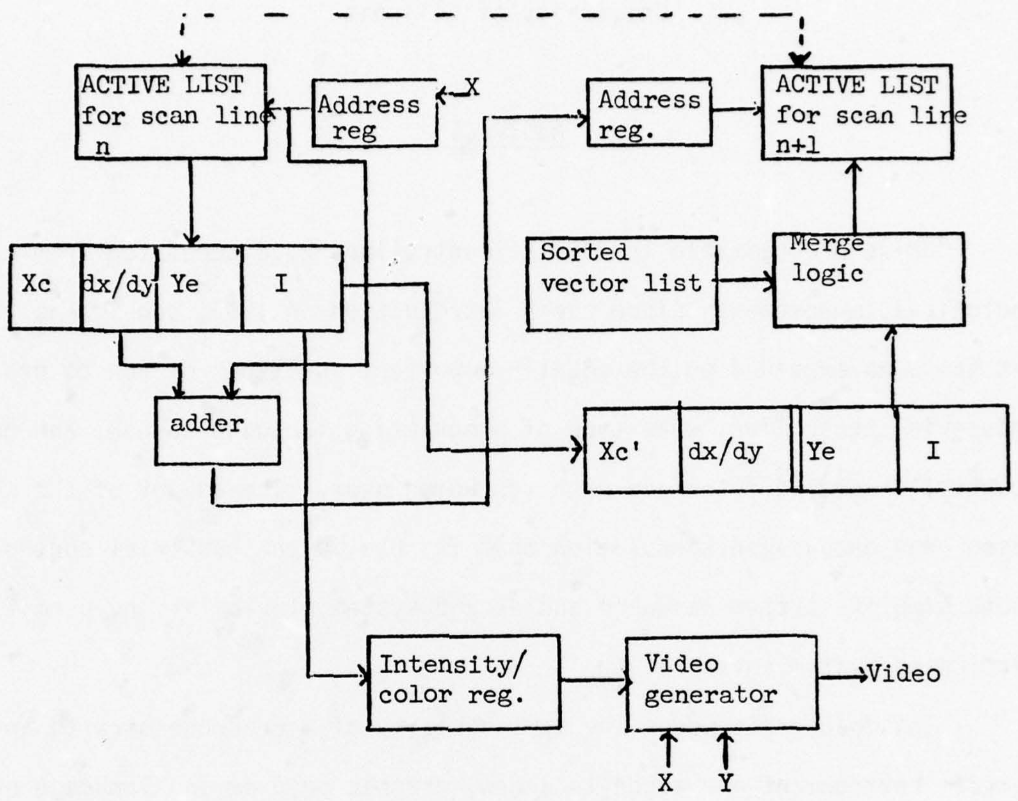


FIGURE 3

# A GRAPHICALLY-PROGRAMMED, MICROPROCESSOR-BASED INDUSTRIAL CONTROLLER

Alfred C. Weaver
Department of Computer Science
University of Illinois

## ABSTRACT

While programmable industrial controllers have benefited from continuous technological improvements since their introduction in 1969, considerably less effort has been expended on the equally important questions of how to program an industrial controller, what type of programming language to use, and how the controller should interface with its human user. The advent of the microprocessor has encouraged speculation that its use in an industrial controller can both simplify system hardware and expand system flexibility by permitting sophisticated system software.

This paper discusses the applicability of microprocessors to an industrial environment and suggests a new, graphic programming language based on familiar relay symbols as the system's input. A design overview is presented for two stand-alone, microprocessor-based machines--the controller itself, which interprets a user program and manages system I/O, and an auxiliary program loader which supervises interactive graphic programming using a special keyboard and CRT. When connected, the two units communicate to provide the user with the capability of monitoring and changing a running system.

## 1. INDUSTRIAL PROCESS CONTROLLERS

The process control systems of the early 1960s were essentially all-relay systems and suffered from a number of critical deficiencies:

1. Each new application meant a new, custom design;
2. Each new design implied expensive field wiring;
3. Hardware costs were high;
4. Relay hardware was unreliable without extensive field maintenance; and
5. Systems, once installed, offered no flexibility for changing or upgrading the control mechanism.

The development of the microprocessor, coupled with the adaptation of computer science software technology, is bringing forth a new generation of industrial controllers which substitutes programming for field wiring and CRT debug monitors for continuity testers.

Programmable controllers are useful in a host of industrial applications. Some typical applications include:

| | | |
|---|---|---|
| conveying systems | sorting mechanisms | packaging operations |
| pumping stations | assembly operations | production testing |
| labeling | transfer lines | batch sequencing |
| machine tools | welding controls | baggage routing |

While the major thrust here is the design of the system software for such a controller (determination of an effective programming language and design of two operating systems, a graphic translator, and a communications package between dual processors), a successful implementation requires that due consideration be given to the hardware environment in which the software is to operate. By designing both hardware and software simultaneously, we have attempted to achieve the proper balance between functions implemented in hardware and those implemented in software.

## 2. CONTROL SEQUENCE SPECIFICATION

In the earlier industrial controllers the decision-making logic functions were performed with either electro-mechanical relays or custom, hard-wired, static logic devices. The documentation for the control function of such systems was the electrical drawing defining how the relays and/or logic devices were to be interconnected to form the control sequence. This "team" of relays and control diagrams has been used successfully for many years. The control diagram which defines the relationships among the input, output, and control variables is called a "relay ladder diagram" (RLD) due to its characteristic symbols and format [1,2].

Described briefly, the RLD uses the leftmost vertical line to represent a source of electrical power ("power line"); current propogates through the matrix as relays enable or inhibit current flow along the horizontal paths. If current does reach an output (indicated by a circle) in the rightmost column the output will turn on, else it will turn off. See Figure 1, a sample RLD, and Figure 2, a legend for some standard symbols.

## 3. PROGRAMMING WITH THE CRT

The "program" is entered via pushbuttons on a custom keyboard and displayed on a CRT. The individual "pages" of an RLD are defined to be a programming matrix of four rows and eight columns. Each matrix position in the first seven columns may contain a space, horizontal connection, normally open relay, or normally closed relay. A vertical connection to the line below may optionally be added to any symbol on line one, two, or three. The relay element addresses may refer to any input, output, control variable, latch, timer coil, counter coil, or any other internal variable; thus, every variable in the system may potentially participate in the control of any output. Column eight is reserved for outputs (normal or complemented, retentive or nonretentive, timer/counter coils, etc.). Any page may contain one to four outputs, one per line. There are no formatting restrictions with regard to the interconnections on a single page.

The program is entered interactively using a blinking screen cursor to identify the matrix position currently being programmed or edited. To insert a symbol in any location, the cursor is simply moved into the desired position by depressing the cursor control pushbuttons:

| | |
|---|---|
| $\uparrow$ | move up one line |
| $\downarrow$ | move down one line |
| $\leftarrow$ | move left one element |
| $\rightarrow$ | move right one element |

Having selected a matrix position with the cursor, any of the relay symbols may now be inserted. The I/O address of the element (the physical terminal number to which that input or output is attached) is entered on a keypad containing the digits 0 to 9. Each address digit entered shifts the current 3-digit element address field (initially 000) one digit to the left and inserts the just-depressed digit in the least-significant digit position. Depressing any of the output symbols automatically advances the cursor to column eight, filling intervening blank elements with horizontal connections, and inserts the output symbol.

If an error is made while programming a given page, the user need only reposition the cursor to the matrix position in error and press the symbol and/or address keys corresponding to the desired correction. In this manner the user may program any matrix position, in any order, as well as edit any number of positions in any order. Any symbol may be added, deleted, or altered with only a few keypresses. After the page has been visually inspected, depressing NEXT PAGE in program mode saves the current page and displays a blank next page; in edit mode it replaces the saved page with the page just edited and then displays the next page as recalled from memory. For ease of editing, accuracy of documentation, and enhancement of visual verification of correctness, the translation technique preserves the exact topology of the input RLD so that it can be reconstructed exactly from the internal code at any later time.

When programming a timer or counter, the user may choose to insert his 4-digit presets/setpoints directly through the keypad. All presets/setpoints may be changed dynamically at any time during program execution for system tuning or correction.

When all the pages of a program have been completed, the COMPILE pushbutton is depressed. The graphic translator software package is then invoked which translates the various graphic pages into internal code.

The utility of the programming format is illustrated by the simplicity of its rules:

1. There are four rows and eight columns per RLD page;
2. Column eight is reserved for outputs;
3. Verticals may not extend below line four; and
4. Lines which cross are assumed to connect.

The user should appreciate the fact that it is impossible to introduce a permanent syntax error. Not only is the language designed for freedom of expression, but the only possible syntax errors are detected by the graphic translator:

1. Nonoutput type symbol in column eight;
2. Vertical entered on line four; and
3. Invalid I/O address for relay element.

In each case the invalid keypress is immediately rejected and an audible beep (error signal) emitted.

The switch from program to edit mode is painless and immediate. Since the system decompiles compiled code exactly, the user experiences no disturbing format change when he edits a program. To further aid his editing, a SEARCH function is provided in which a relay symbol and address are given. Each sequential depression of the SEARCH key locates and displays the next page of RLD code in which the given symbol is used in conjunction with the given address.

Other modes, including MONITOR, FORCE, and MEMORY COPY, allow the user to interactively debug a running program by watching relays conduct and outputs close in real time, or by forcing any input on or off. Once a complete RLD has been entered, it is compiled and stored in a 1Kx8 RAM within the program loader. After further execution, testing, and possible alteration in RAM, the program may be permanently stored by inserting a 1Kx8 PROM and depressing a "copy RAM" key; the PROM will be automatically programmed to duplicate the content of the RAM. Likewise, a previously compiled program stored in PROM may be transferred to RAM by depressing a "copy PROM" key; the program will be decompiled and made ready for display, monitoring, and/or editing. Programs in RAM and PROM may be verified to be identical by use of the VERIFY key; a CRT message reports the address and content of any mismatch.

In summary, the graphic programming "langauge" described provides numerous benefits over any other programming system currently available. Its use of standard relay symbols, keypress programming, easy program entry and editing, and reliance on visual verification of correctness allow the user to capitalize on his intuition; drawing his RLD on the CRT is no more trouble than drawing it on paper.

From the user's point of view, the most important advantages accrue from the unrestricted format of the RLD input (unique to the industry), the ability to recreate the exact RLD used as input from the internal code alone (also unique), and the freedom to define up to four outputs per relay page (likewise unique).

## 4. LANGUAGE TRANSLATION

Having defined an acceptable graphic programming language, a significant problem still remains--how to translate the RLD into an internal code in such a way as to extract its Boolean content while preserving its topology. Historically, the imposition of formatting restrictions has served the purpose of reducing the complexity of the translation process. But now we search for a suitable translation scheme which will neither impose format restrictions nor be so complicated as to strain the space and speed limitations of a microprocessor-based system.

Into what should the RLD be translated? The two basic alternatives are to produce executable code for the chosen microprocessor or to produce an intermediate text which can be interpreted. Although the former offers the advantage of high-speed execution, the latter scheme will conserve memory space as well as simplify conversion to a different microprocessor in some future version of the hardware.

The compilation of a graphic RLD into directly executable microprocessor code may be accomplished by Boolean equation templates or recursive maze traversal, producing either true Boolean equations or their parse tree, as desired [3,4,5]. The direct advantage of either technique is that both the compilation and optimization techniques are well known [6,7,8] and could be expected to produce high quality microprocessor code. On the other hand, the number of terms in the equation and the complexity of the optimization pass would require a large amount of microprocessor memory for both the translation program (in ROM) and the working store (in RAM). The time of translation would be extended by the optimization pass, although this is a minor consideration. Most importantly, the optimization of the microprocessor code would prohibit using the code sequence itself to reconstruct the original RLD.

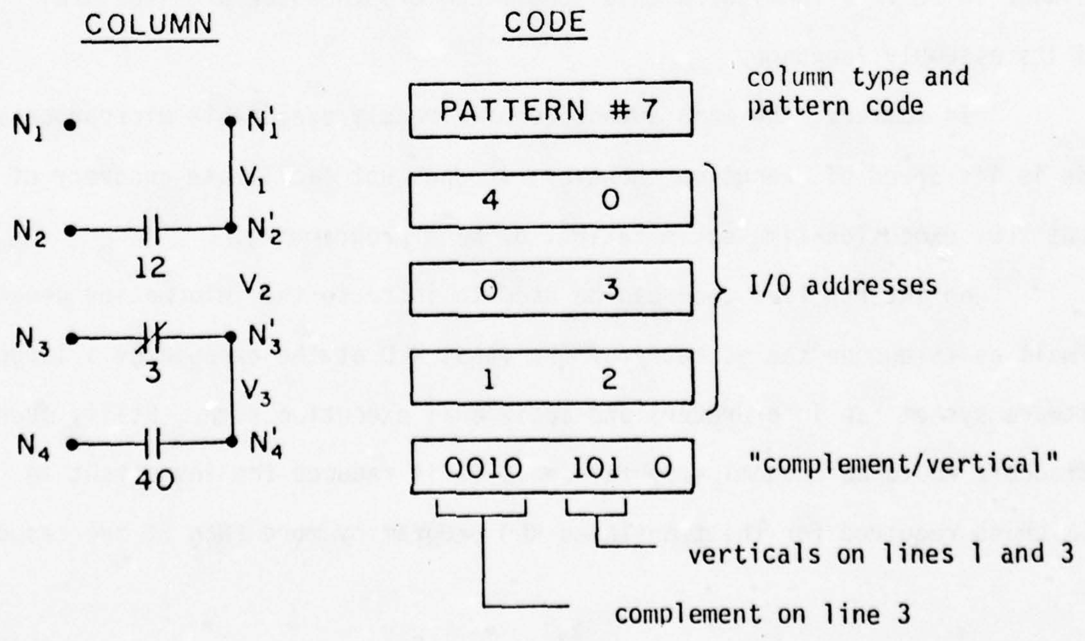
Benchmark programs containing approximately 300 relay symbols have been compiled into directly executable code for the most popular microprocessors (Intel 8080, Motorola 6800, MOS Technology 6502, Zilog Z-80) and, without significant exception, produce approximately 3000 bytes of code with an execution time of about 4 ms. While the execution time is well within the desired range, the memory requirement is not. The information density (memory bytes required per relay symbol) is low due to the bit masking and shifting required by the microprocessor architecture and the overhead of keeping two copies of the output and control variables (necessary to guarantee equation independence).

The excess memory requirement is not the only problem, however. Since the code embodies the logic, but not the topology, of the RLD input, the original geometry cannot be recovered from the microprocessor code alone. Additionally, execution-time optimization, such as recognizing that the evaluation of a lengthy expression is unnecessary if the result is to be ANDed with a zero, is possible only at the expense of even more memory. Further, should the system be programmed by hand, rather than by the graphic translator, using microprocessor code as the source language would require the user to be very familiar with a specific microprocessor architecture and its assembly language.

In summary, the main advantage of directly executable microprocessor code is its speed of execution; however, it does not facilitate recovery of the input RLD, execution-time optimization, or hand programming.

An interpretive code can be used to increase the information density as well as to encode the geometry of the input RLD at the expense of a larger software system (an interpreter) and additional execution time. Still, such a tradeoff would be economically favorable if it reduced the investment in PROM chips required for the translated RLD program by more than it increased the investment in ROM chips required for the whole operating system (assuming execution time remained satisfactory). If the interpretive code is column-oriented and evaluated left-to-right, software can detect rather easily when *current propagation* has ceased and can jump directly to the output column and begin storing outputs. This ability to dynamically skip the evaluation of "non-participating" relay columns recovers some of the time lost to interpretation.

To define an adequate column-oriented code, note that in any one of seven relay columns on a page there can be only 16 positional combinations of relays in any one column; therefore, we define 16 distinct "relay column patterns." The pattern number (0 to 15) identifies the geometry of the column (pattern # 0 indicates no relays, pattern # 1 indicates one relay on line 4, . . ., pattern # 15 indicates one relay on each of lines one, two, three, and four). Additional memory bytes identify the particular I/O address of each relay element in the column. One additional byte indicates which, if any, of the relay elements are complemented (normally closed) and which, if any, of the three possible vertical connections between lines are present. An example follows.

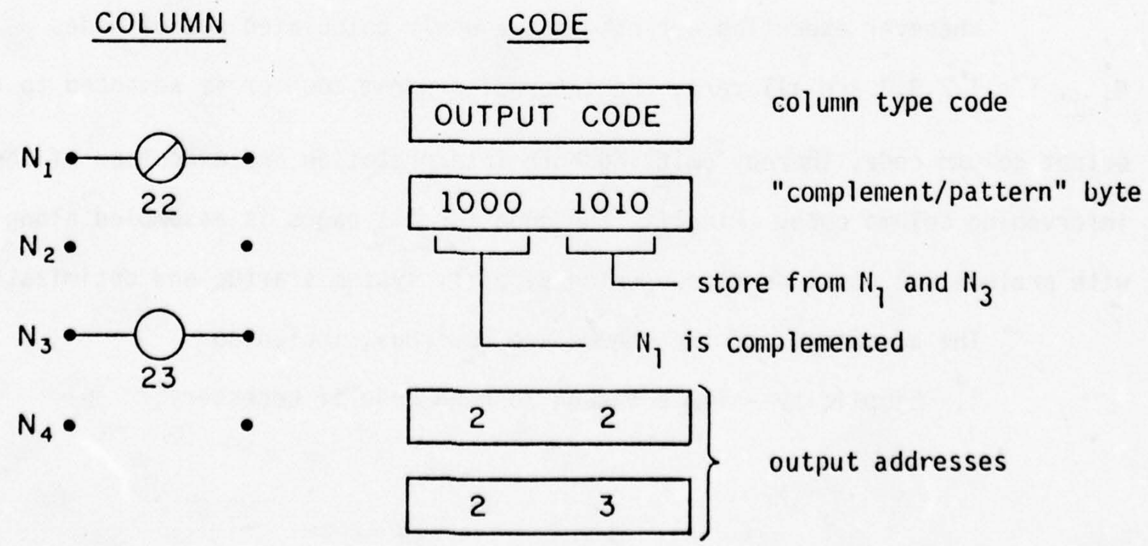


"Executing" the $j^{th}$ column now reduces to fetching the dynamic status of the relay elements of the column, $R_{i,j}$, i = 1,2,3,4, ANDing this 4-bit vector with the left-hand nodes $N_{i,j-1}$, i = 1,2,3,4, and creating a new set of right-hand nodes $N'_{i,j}$, i = 1,2,3,4, after accounting for the influence of complemented variables and additional propagation due to vertical connections. Since the "solution" of the column conduction problem is a function of fifteen variables,

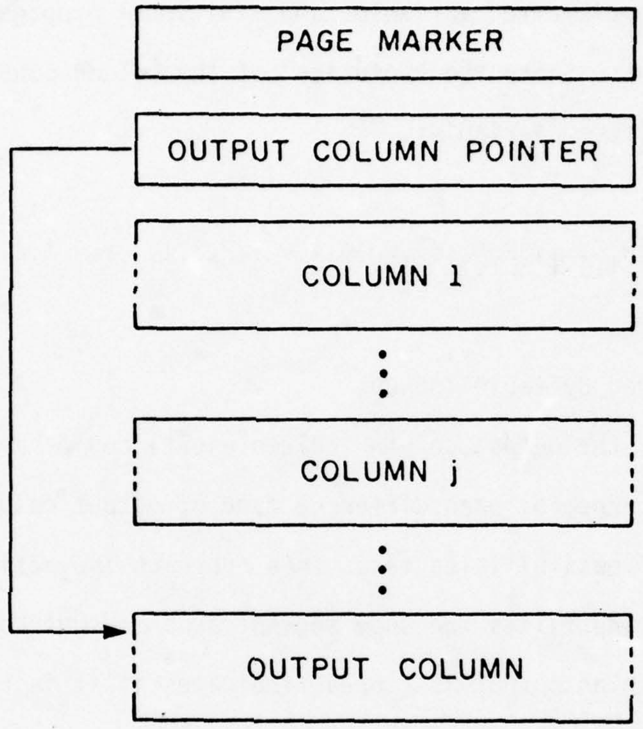
$$N'_{i,j} = f(N_{i,j-1}, R_{i,j}, C_i, V_k) \quad i = 1,2,3,4, \quad k = 1,2,3,$$

it is speedily solved by table lookup.

Likewise, the output column (column eight) could have been segmented into different patterns for each different type of output column possible, but the large number of possibilities makes this approach impractical. Rather, the output column code identifies the code segment as an *output column*, *identifies* the lines from which an output is stored, indicates if it is complemented and to what output address the computed result is to be stored (see below).



The code segments for each page are assembled in column order, left to right, and surrounded by a page marker and a pointer to the output column code as shown below.



Whenever execution detects that a newly calculated set of nodes $N'_{i,j}$, i = 1,2,3,4 are all zero, the internal program counter is advanced to the output column code, thereby omitting both interpretation and execution of the intervening column code. Finally, the code for all pages is assembled along with prelude and postlude blocks which simplify system startup and optimization.

The advantages of the scheme are numerous, including:

1. Simplicity--simple enough to hand code if necessary;

2. Fast--worst case (no optimization branches) for the 300 symbol benchmark program is 25 ms;
3. Economical--a 1Kx8 program PROM (1 chip) will hold a 600 symbol RLD;
4. Input recovery--the exact topology of the input RLD is wholly specified within and recoverable from the internal code; and
5. Moderate software complexity--the operating system for the controller (including the interpreter) is 2K bytes; the operating system, interface, and graphic translator/editor/monitor for the program loader is 8K bytes.

## 5. THE TOTAL SYSTEM

In addition to graphic compilation, the program loader provides complete programming and editing services in a stand-alone, interactive environment. Additional modes of operation allow the loader to be used as an in-the-field monitor. Putting the loader in MONITOR mode allows the user to gain access to a running program in a controller, copy it, decompile it into the exact original geometry and watch relay contact closures displayed in real time on the CRT screen. A FORCE mode permits the user to isolate any number of inputs and outputs and force them on or off to test both programming logic and field connections.

Communication between the program loader and controller is accomplished by an interrupt-driven scheme in which the loader can request status information from the controller and/or override controller calculations. The two independent units communicate through PIAs connected via a single cable. The modular design insures that one program loader may service all of the controllers in any one application, thus reducing total system cost.

Figures 3 and 4 show the physical keyboard, CRT, and mode selection layout of the program loader. Figure 5 shows the hardware required by the communications package.

## 6. IMPACT ON MICROPROCESSOR SYSTEM DESIGN

This work refutes the contention that microprocessors are too impotent to benefit from elaborate software systems (compilers, interpreters, operating systems, etc.). Strict adherence to the principles of structure, isolation, and modularity slowed the initial design and implementation process, but paid off handsomely in the accuracy and reliability of the final product. The development cycle for hardware and software required approximately two man-years each. The most vexing problem area was the implementation of the elaborate asynchronous communications protocol between the two independent processors. It is interesting to note that a subsequent controller and programming unit with four times the processing power of this system was designed and built from scratch in eight man-months each for hardware and software; the communications scheme was not a problem here due to prior experience. The clear implication is that, with a little experience, progressively more elaborate software systems will be permitted in a previously hardware-dominated domain.

One should not infer, however, that current microprocessor architecture is at its zenith. Rather, the expanding use of microprocessors is sure to have a significant impact on future microprocessor design. While currently available microprocessors tend to follow the conventional architectures of previous mini- and maxi-computers, increased usage in special purpose applications will justify the design of new, special purpose microarchitectures. Although the solution of relay ladder diagrams is only one instance of a special purpose application, it nevertheless serves to illustrate the limitations of existing microprocessors and to suggest improvements in future products. In the area of industrial process control, five observations can be made.

Processor speed is marginal.

Instruction times of 2-5 microseconds generate unacceptable delays when attempting replacement of conventional hardware logic elements with their software equivalent. As software systems become larger and more complex, faster processors (perhaps of bipolar technology) will be mandatory. Until then, discrete outboard logic (e.g., PLAs, miltiplier units) will be used to augment standard microprocessor capabilities.

Eight-bit data paths are too restrictive.

Process control deals with Boolean control (1 bit), BCD digits (4 bits), ASCII characters and status vectors (8 bits), binary counters and internal addresses (16 bits), and data transfer (typically 1K bits). Few microprocessor instruction sets are as effective with bits as they are with bytes; as a result, excessive bit-masking, shifting, and packing reduce efficiency. An instruction set which effectively handles variable length data (i.e., bit, byte, word, and block) would be a distinct advantage.

Instruction sets must provide multiple addressing modes.

Intel chose to feature a diversity of instruction types; Motorola chose to implement fewer instructions with more addressing modes. The latter is preferable. The system software of the controller alone used every addressing mode in the 6502 and could have benefitted from even more elaborate ones (double indirect, double indexed).

We need more elaborate software support.

Manufacturer-supplied development software varies widely in its quality. Assemblers are well understood and are generally adequate; simulators are not; compilers are a joke. The productivity and efficiency of microprocessor system designers are proportional to the quality of software support systems used. While PL/M-type compilers are useful for initial design evaluation and debugging, their code is too inefficient for use in production environment.

Microprogrammable architectures will blossom.

The flexibility of defining system architecture via microcode is extremely advantageous. Microprogramming allows the designer to generate exactly those instructions and addressing modes deemed most useful for the application. The cost, however, in design time, prototyping, and system debug time (both hardware and software) is enormous. This situation is due to a general industry unfamiliarity with microprogramming concepts and, again, inadequate hardware and software tools for generating and testing microprograms. Once support is established, microprogrammed processors will be the way of the future.

## REFERENCES

[1] "Graphic Symbols for Electrical and Electronics Diagrams," Standard Y32.2, United States of America Standards Institute, New York, NY 10016.

[2] "Standard IC-1, Industrial Control," National Electrical Manufacturers Association, New York, NY 10017.

[3] Weaver, Alfred C., VIPTRAN - A Programming Langauge and its Compiler for Boolean Systems of Process Control Equations, M.S. thesis, Department of Computer Science Report No. UIUCDCS-R-73-603, University of Illinois, Urbana, IL 61801, November 1973.

[4] Weaver, Alfred C., VIPTRAN2 - An Improved Programming Language and its Compiler for Process Control Equations, Department of Computer Science Report No. UIUCDCS-R-74-643, University of Illinois, Urbana, IL 61801, May 1974.

[5] Weaver, Alfred C., A Graphically-programmed, Microprocessor-based Industrial Controller, Department of Computer Science Report No. UIUCDCS-R-77-865, University of Illinois, Urbana, IL 61801, May 1977.

[6] Aho, Afred V., and Ullman, Jeffrey D., The Theory of Parsing, Translation, and Compiling, Vol. 1, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.

[7] Hopgood, F.R.A., Compiling Techniques, American Elsevier, New York, NY 10017, 1969.

[8] Gries, David, Compiler Construction for Digital Computers, John Wiley & Sons, Inc., New York, NY, 1971.

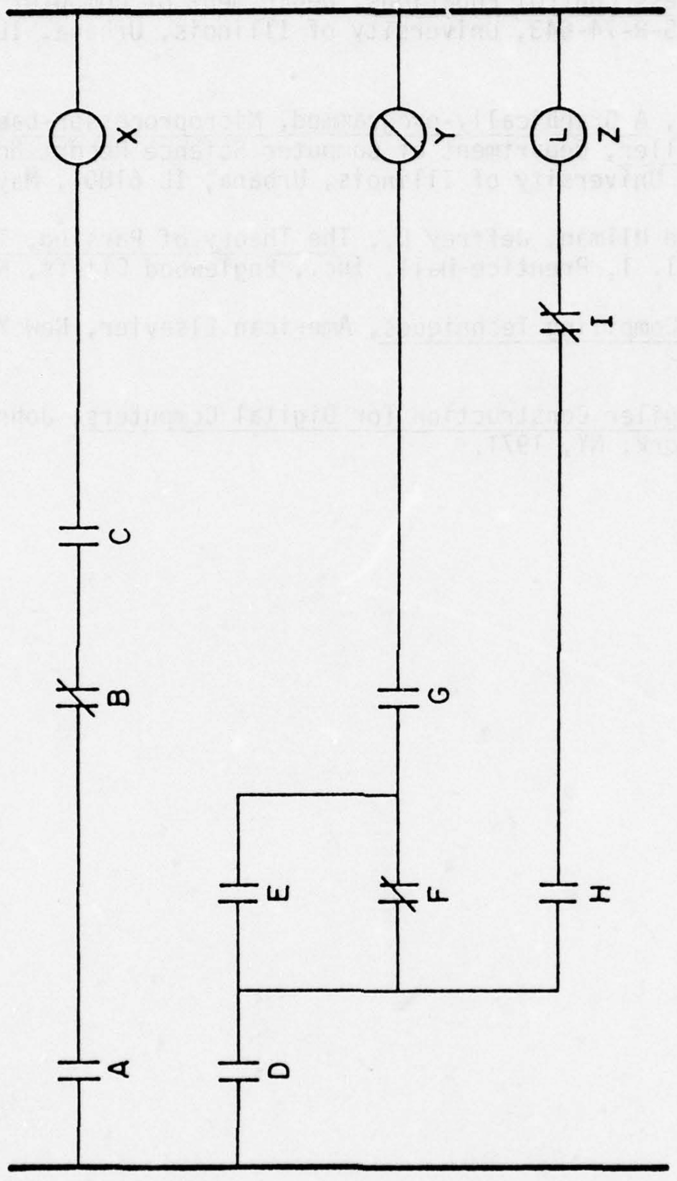


Figure 1. Relay Ladder Diagram

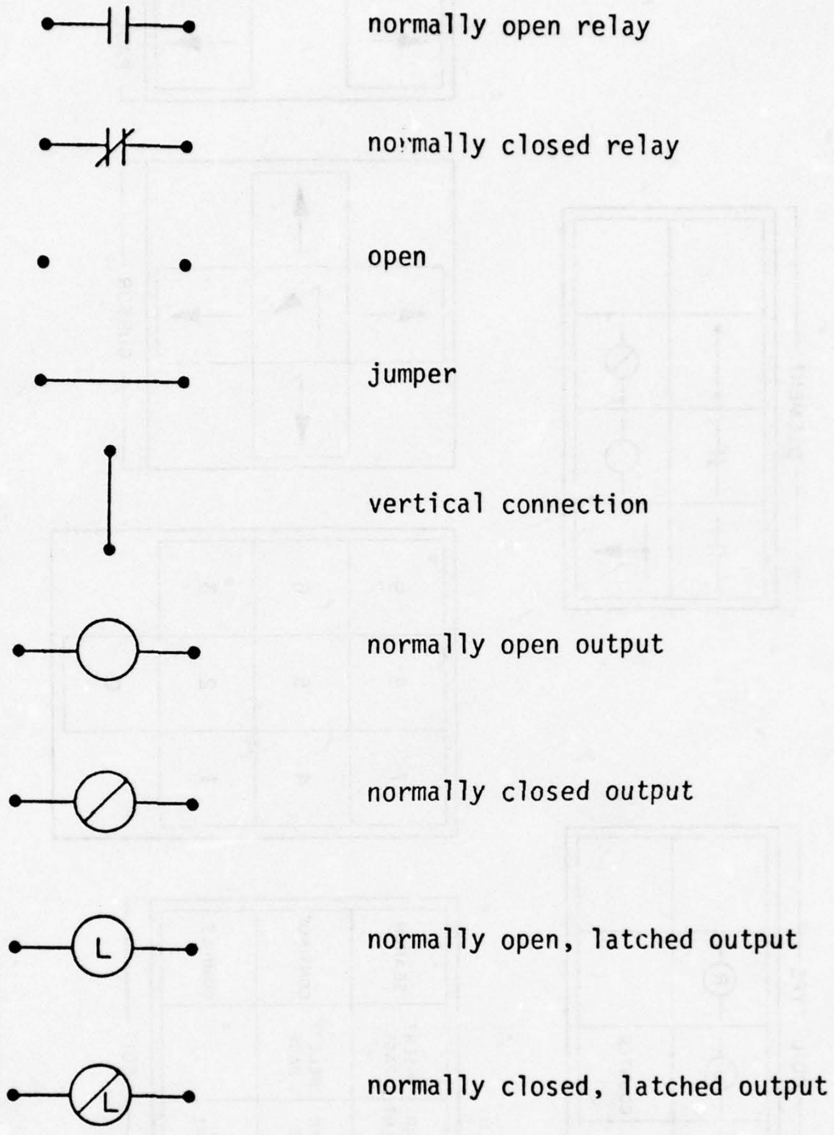


Figure 2. Some RLD Symbols

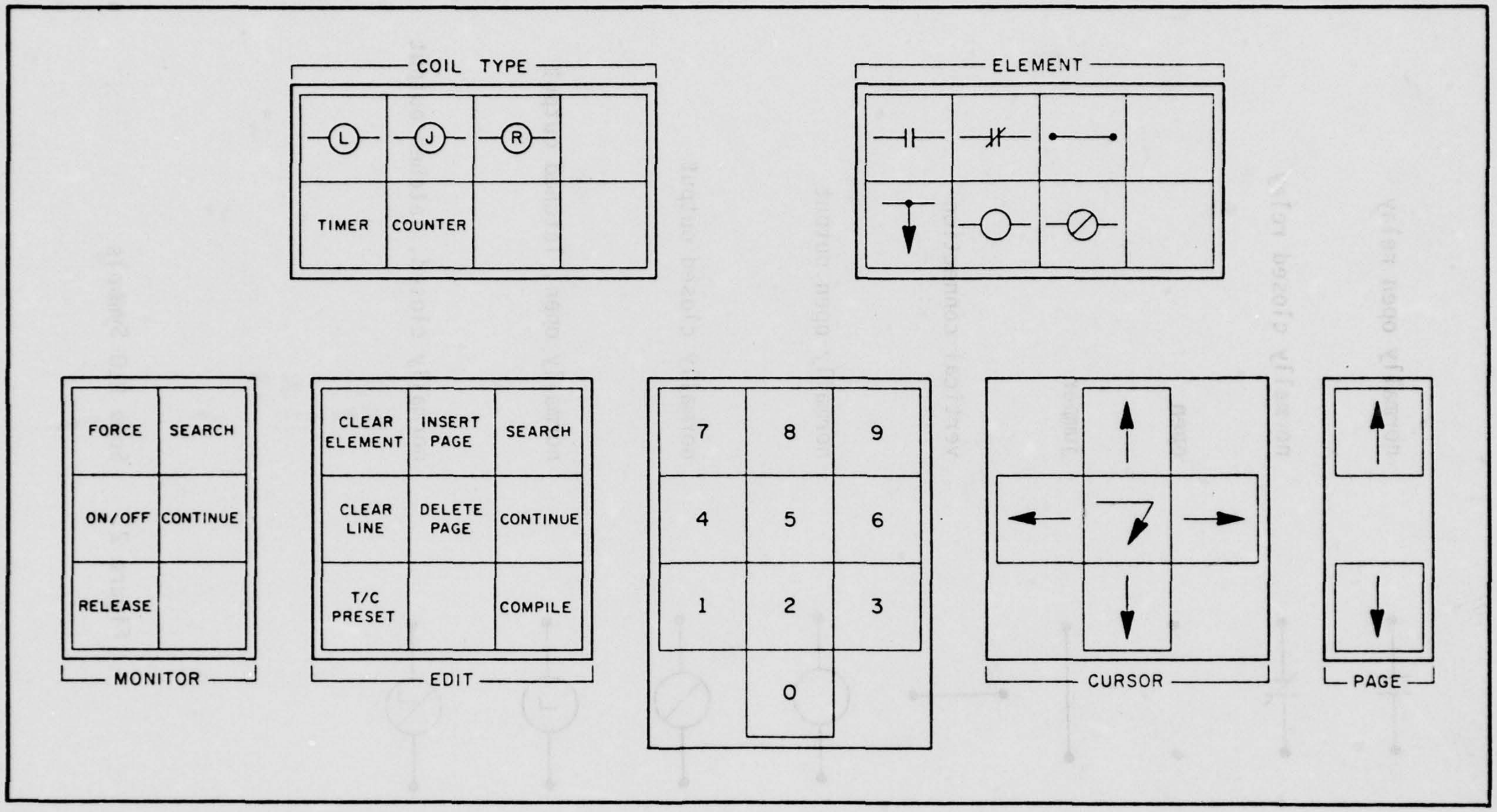


Figure 3. Programming Keyboard

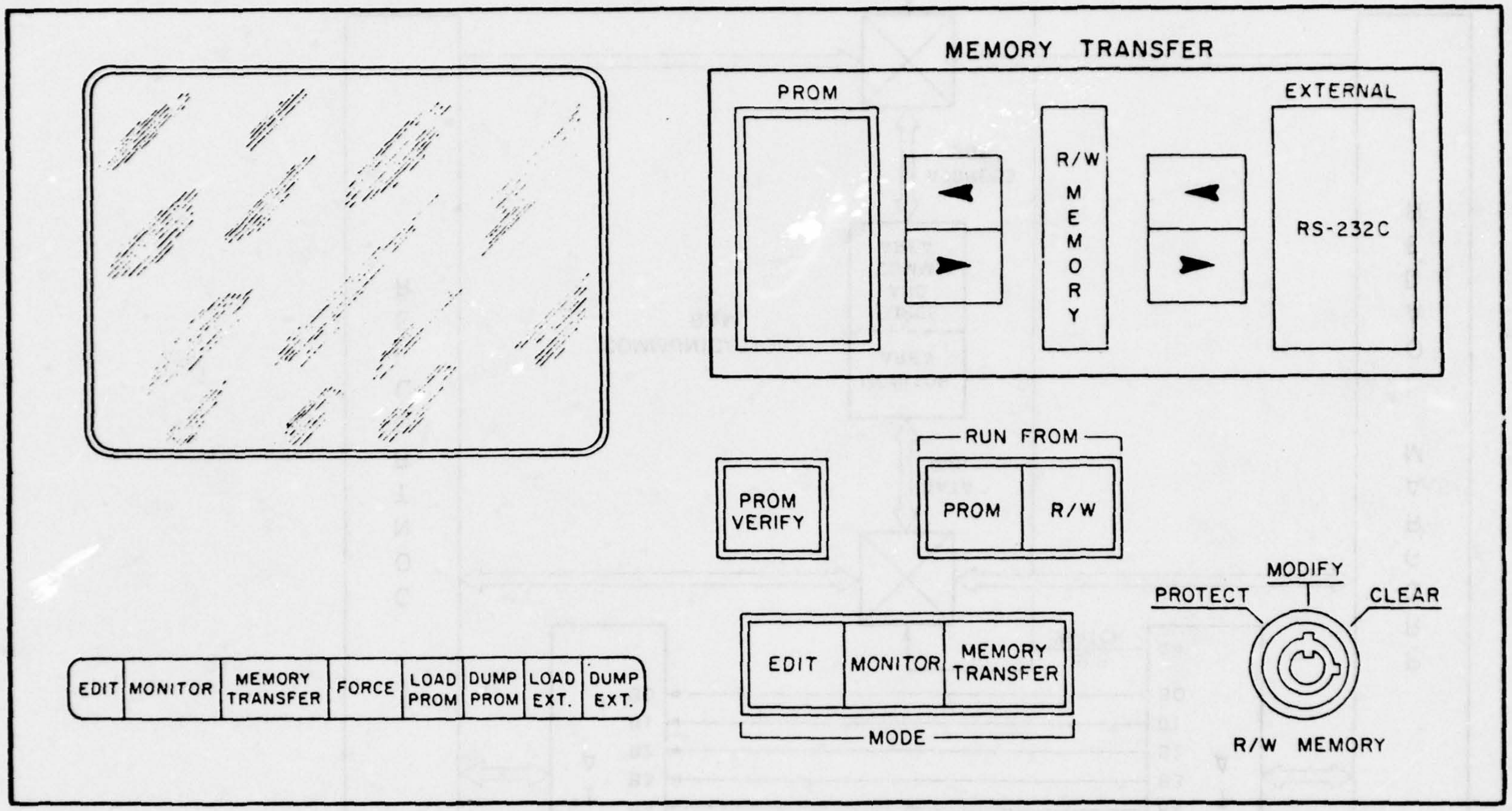


Figure 4. CRT and Mode Selection Keyboard

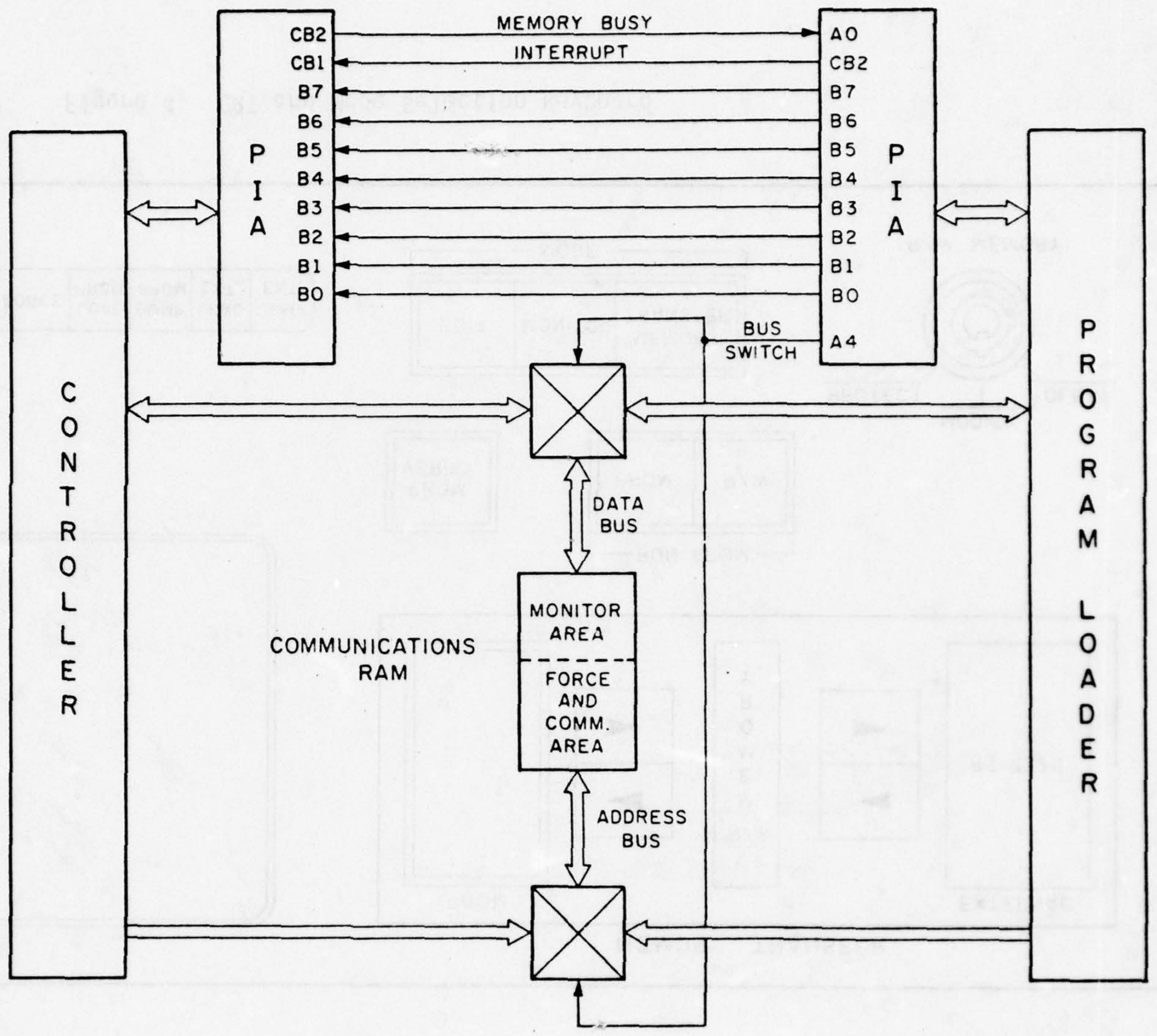


Figure 5. Communications Hardware

# MIDAS - A MICROPROGRAMMABLE INTEGRATED DATA ACQUISITION SYSTEM

James W. Sturges, Thomas M. Houlihan
Naval Postgraduate School
Monterey, California

Ronald A. Hess
NASA/Ames Research Center
Sunnyvale, California

## ABSTRACT

A Microprogrammable Integrated Data Acquisition System (MIDAS) was developed as a compact, lightweight, and economical system for the acquisition of medium-speed analog signals. The system contains a microcomputer, an incremental digital cassette tape recorder, a high-speed 16-channel multiplexed analog-to-digital converter, a digital clock, and a keyboard for program or parameter entry. The details of operation and programming are described.

## I. INTRODUCTION

The research and development effort whose product is described in this work was undertaken in an attempt to produce an alternative to the large, cumbersome, and expensive analog tape recorder. A Microprogrammable Integrated Data Acquisition System (MIDAS) was the result.

Quite often, the analog tape recorder provides frequency response and data volume not required for a specific investigation. This implies that an optimum equipment suite is not always the most elegant. On the contrary, the optimal suite carefully mates enough samples for the resolution required. With the analog tape recorder, data are presented as a continuum, and must be discretized.

An initial attempt, using discrete logic devices, although feasible, had limited utility because of its hard-wired configuration. It did provide excellent speed and reliability in taking data samples at a fixed rate from a fixed number of analog devices and recording digitized values on magnetic tape. Although this was a substitute for the analog recorder, it did not provide the flexibility which was desired, since it ran like the analog machine at a limited number of speeds, and was constrained by hard-wiring to do only one job.

In order to provide a large degree of flexibility, a microcomputer was used to replace the hard-wired discrete logic devices. The microcomputer gave to MIDAS the same capabilities which a maxi- or mini-computer would give, with a tolerable penalty in speed and memory capacity. Programs could be written in high-level or assembly language to replace the functions once performed by discrete logic devices. Thus, MIDAS was capable of sampling a variable number of channels a variable number of times (variable among the channels), upon some variable external command or sequence of events.

The system combined a high-speed analog-to-digital converter, an incremental digital tape recorder, and a second-generation microcomputer in a compact, lightweight package. Requiring only external electrical power, the system was capable of operation in an infinite number of modes. The completed product cost less than three thousand dollars and has been used to replace analog tape recorders costing five times that much in cases where the data resolution could be handled by MIDAS.

## II. SYSTEM DESCRIPTION

### A. PRIMARY COMPONENTS

MIDAS used three commercially available units from original equipment manufacturers, each of which made a large contribution to the entire MIDAS system. The three units were: a DATEL, Inc., model DAS-16 multiplexed modular data acquisition system; a MEMODYNE, Inc., model BR-103 incremental digital cassette tape recorder; and the heart of the system, a PRO-LOG, Inc., model MPS-805 microcomputer.

1. Modular Data Acquisition System

The DATEL DAS-16 incorporated a series pair of eight-channel analog multiplexers, a sample-hold unit, a high-speed analog-to-digital converter (ADC), and a system programmer module, in a 4.5 inch by 5.0 inch module only 1.5 inches thick. The particular series of DAS-16 used in MIDAS was capable of 25 Khz throughput, and had a resolution capability of 4096 counts over a ±5 volt range.

The DAS-16 could be operated in either random address or sequential address mode. In the random address mode, the DAS-16 was forced to the addressed channel, and had to remain on that channel until the address command was changed. In the sequential mode, the system programmer module was pulsed with a CONVERT* command, and it in turn advanced the multiplexer address to the next sequential channel when the conversion was complete. The RESET* command was used to drive the system programmer to output the address for channel 1 to the multiplexer, and operated in either address mode. Although provision was made for wiring the DAS-16 to reset upon reaching a pre-defined upper channel address, the capability was not used in MIDAS.

## 2. Incremental Tape Recorder

The MEMODYNE model BR-103 tape recorder was configured to use the industry standard "PHILLIPS" digital tape cassette, a certified computer-grade tape cassette. In the MIDAS system, this cassette was capable of storing over 1.7 megabits of information. Data were presented to the recorder in bit-serial format, with both MOTOR CLOCK and WRITE CLOCK waveforms generated by the microcomputer in the system. The recorder used a two-track recording system such that a flux change on one track signifies a logical "1", while a flux change on the other track signifies a logical "0". MEMODYNE provides both a "sync" output and input for signaling end of byte, although the sync system was not used in MIDAS. There were four circuit cards used in the tape recorder, three of which were purchased from the manufacturer, and one which was manufactured locally.

### a. Motor Drive Card

The Motor Drive Card received a waveform called "MOTOR CLOCK," which triggered a one-shot. The output of the one-shot triggered a flip-flop connected to the motor drive amplifier circuitry, such that the resulting output to the motor was a series of steps applied to alternate windings. The MOTOR CLOCK waveform was generated by a series of trigger pulses from the microcomputer at programmably definable intervals. The logic which would be required to control an external oscillator during write operations, in a dedicated hardware system, was replaced by program statements executed by the microcomputer. Although the card had provision for operating the motor in reverse, the provision was not used in the MIDAS system.

b. Write Step Card

The Write Step Card received both the SERIAL DATA and the WRITE CLOCK waveforms, and generated the pulses which were used to alter the magnetic signature on the tape. The WRITE/READ* line was also used in this card to determine whether the machine was in the WRITE or READ status. When a WRITE CLOCK pulse was received, the data presented was used to write a flux change on the appropriate track.

c. Read Amplifier Card

The Read Amplifier Card sensed the flux changes on the tape during READ operations, and produced two signals used by MIDAS. The TAPE CLOCK and TRACK 1 DATA signals reproduced the waveforms presented to the WRITE CLOCK and SERIAL DATA inputs on the Write Step card.

d. Read Oscillator Card

The Read Oscillator Card, manufactured locally, was used to produce a MOTOR CLOCK waveform when the recorder was used in the READ mode. This card used a SIGNETICS NE-555 Timer integrated circuit, which could be adjusted to run at frequencies around 360 Hz, the recommended motor stepping frequency during READ operations. The oscillator was gated ON by driving the READ* line to logical "0", and the MOTOR

CLOCK line to logical "1". Since the recorder was never in this condition in the WRITE mode, this logical state implied that the system microcomputer was directing a READ operation.

The necessity for the separate oscillator card is not at once obvious, since the microcomputer can generate the required waveforms for both the MOTOR CLOCK and the WRITE CLOCK during WRITE operations. It must be remembered, however, that the signals on the tape may appear in a pseudo-random manner, due to vibration of the recorder while writing, uneven tape slack, etc. If the microcomputer were required to produce the MOTOR CLOCK waveforms, it is possible that a TAPE CLOCK pulse might pass undetected during the MOTOR CLOCK pulse generation routine sequence.

### 3. Microcomputer

The microcomputer selection for MIDAS was determined by the microprocessor used in the microcomputer. Although there existed a plentiful number of discrete microprocessor devices, only one company had to date, supported its microprocessor with a higher-level-language. That company was INTEL, Inc., which provided the PL/1 - based language, PL/M, especially written for two of its microprocessors, the 8008 (used in MIDAS), and a new version, the 8080. Since the question of the microprocessor had been decided, it was then necessary to search for a manufacturer who included the 8008 or the 8080 in a microcomputer. Although INTEL manufactured prototyping systems for both of their devices, they were much too bulky for inclusion in a working system. Additionally, a rather large price differential existed between INTEL and their only competition, PRO-LOG. Since the PRO-LOG machine was inexpensive, lightweight and compact, it was selected.

The MPS-805 microcomputer used five card modules: a Central Processor Unit (CPU) card, a Read Only Memory (ROM) card, a Random Access Memory (RAM) card, an Input card, and an Output card.

## B. SECONDARY COMPONENTS

Included in the MIDAS package were a digital clock, with time readout available to the microcomputer input, and a keyboard which was used to enter execution instructions, parameters requested by programs currently executing, and new program instructions.

A locally manufactured card, the Digital Input Select Card (DISC), was used to multiplex the microcomputer's input ports. Also, four HEWLETT-PACKARD integrated circuit hexadecimal displays were installed in the MIDAS front panel.

### 1. <u>Digital Clock</u>

The digital clock used in MIDAS was MM-5313 integrated circuit. The clock required a 60 Hz input, and provided a multiplexed binary output as well as multiplexed outputs for driving seven-segment numeric displays. The 60 Hz signal was derived from the microcomputer quartz crystal timebase which provides timing signals to the microcomputer. A "hold" was provided for time setting, in conjunction with a "fast" pushbutton and a "slow" pushbutton, for advancing the displayed time to the proper time in fast or slow fashion. The outputs from the clock were sent to a DISC card, and could be read into the microcomputer on command. Six seven-segment bar displays displayed hours, minutes, and seconds on the front panel.

2. Keyboard

The keyboard originally designed for a desk calculator, was modified to allow it to output all the hexadecimal characters. The modification consisted of changing the special purpose keys for mathematical operations to the keys necessary to output the characters not shared between the decimal and the hexadecimal counting systems (i.e., A, B, C, D, E, F). The multiplication key was left connected as a special function key, and was used for "word entry", signifying that a byte is ready for input to the microcomputer.

3. Digital Input Select Card (DISC)

The DISC was used for multiplexing up to four inputs per card into a common input port. Using "wired and," or open collector output NAND gates for data selection, the DISC received an address (1-4), and output data presented to its corresponding input port.

Any number of DISCs could be bussed into the same input port, as long as the DISC which was active was not in competition with another DISC on the same bus.

4. Hexadecimal Display

Four HEWLETT-PACKARD hexadecimal displays, mounted in the MIDAS front panel, were used for verification of memory content, operator prompting, etc. The displays were configured as five columns of seven rows of light-emitting-diodes.

## III. SYSTEM OPERATION

### A. GENERAL

As emphasized before, MIDAS operated as a software system. The mesh between understanding the software and operating the system is so tightly woven that even the barest understanding of MIDAS operation depends on understanding the program structure. With that premise in mind, software modules were written and de-bugged. Since these modules existed in subroutine form, in ROM, they could be linked together and executed with a minimum of user-written code.

### B. MICROCOMPUTER MEMORY ORGANIZATION

The MPS-805 microcomputer's memory was arranged as a set of pages, each with 256 lines. Each line was a byte of eight bits, which could be represented as a pair of hexadecimal characters. There were three pages of ROM and twelve pages of RAM installed in MIDAS for general data collection. Two of the general-purpose registers, H and L, were used for "pointing" at a particular place in memory, either ROM or RAM. Register H was used to point to the page desired, and register L points to the line desired.

## C. SOFTWARE MODULES

In MIDAS, all of the software modules were written as subroutines. It was therefore possible to link one or more of these subroutines into a working data acquisition package. A listing of the primary subroutines, together with a brief description of each, is presented below.

### 1. <u>EXEC</u>

The EXEC routine was accessed by the computer at any time the RESET button was pressed, or whenever power was restored to the system. The machine code for a JUMP UNCONDITIONAL instruction was loaded into the first line in RAM; then, using subroutine KEY, the keyboard was read. The user keyed in the line number, and then the page number of the location to which he wished to jump. The JUMP UNCONDITIONAL instruction transferred complete control to the program located at the addressed location.

### 2. <u>KEY</u>

The KEY routine read the keyboard a character at a time. When a key was depressed, after KEY had been called, the value indicated on the key was transmitted to input port 0.

### 3. <u>TIM 1</u>

TIM 1 provided a delay in increments of 11.12 milliseconds, used in the KEY routine for de-bouncing the keyboard switches. The number of delays was loaded into register D, and TIM 1 was called.

4. RCDR

RCDR operated the cassette recorder. By generating the MOTOR CLOCK as well as the WRITE CLOCK waveforms, the microcomputer had complete control over the recorder. Since it was desirable to minimize code whenever possible, RCDR was used to generate "gaps" where only the erase head was in operation, and no flux reversals were written onto the tape, as well as when normal write operations were desired.

5. LGAP

LGAP used RCDR to write a long gap on the tape. LGAP was used to erase a markedly long section of tape, or to make a long leader at the beginning of a new cassette.

6. SGAP

SGAP wrote a gap equivalent to four eight-bit bytes, and was used to place the tape in motion prior to writing on the tape.

7. HDR

HDR used routines KEY and CLK to produce a "standard" header of length 10 bytes, beginning at the first line of RAM. Using routine CLK, the hours and minutes output from the clock were loaded and the routine then called KEY to load values into locations.

8. DEMO

DEMO is a demonstration of module linkage. This routine called LGAP, HDR, DGTR, and returned to HDR when the first data acquisition/record sequence was complete.

9. DGTR

DGTR represented the core of the software. Using information placed into RAM by HDR, DGTR selected analog signals by sequencing through the DAS-16 and stored their digitized equivalents in RAM until a variable (user-defined) length buffer was filled. When the buffer was filled, DGTR called RCDR to record the buffer. If there were remaining data points to be taken, DGTR continued until the number of pages of data points specificed by the user had been satisfied.

10. CLBTR

CLBTR was used to calibrate the DAS-16, to examine a relatively steady-state analog signal, or to set the zero and full-scale points on an input. A "99" appeared in display 0 when CLBTR was called, and the user entered the hexadecimal code for the channel he wished to examine.

11. READR

READR was used to read from the tape cassette, and could be used for either data retrieval or program entry into the machine. The user keyed in a line number and page number where the tape information was to be stored. The user then keyed in the total number of bytes to be read in (up to 256), in hexadecimal; READR then read that many bytes into the memory.

12. <u>AUDR</u>

AUDR was a dual-purpose routine, used for both loading RAM and examining ROM or RAM. After accessing the routine, the user keyed a "1" for loading, or a "0" for examining, followed by the line and page number of the code he wished to load or audit. Port 0 display showed the line number to be operated on, in load, or the line number of the data presented in port 1. In examine mode, the KEY routine was used only as an event monitor. In the load mode, the KEY routine actually loaded RAM.

## D. OPERATING INSTRUCTIONS

After applying power, the user observes both display ports showing "00". This condition exists at any time the machine is reset or powered up. The machine is executing EXEC, waiting for a two-part address (line and page number) to which to jump. The user then keys in the line number, followed by "word entry," and the page number, followed by "word entry." If, for example, it is desired to use DEMO, the user will key "FO," "word entry," "word entry" (the "word entry" key by itself is an implied zero). If a tape is loaded, and the head is down, the tape will begin to move, and port 0 will count down from "07" to "00." Almost immediately, a "01" will appear in port 0, prompting the month. The user then keys in the number corresponding to the month, followed by "word entry." The display will then prompt "02," calling for the day, which should be keyed in, and then the display will prompt "03," which should be answered with the year. The day, month, and year will be stored as pairs of binary-coded-decimal numbers, rather than as hexadecimal numbers. This is the only information keyed in as decimal numbers. The "04" prompt calls for the total number of pages (256 data points) to be taken. With 11 pages of RAM installed, the user's answer may be in the range of 1 to B (remember, hexadecimal input). When prompting "05", the machine is asking for the highest channel number to be included in the scan. The range of answers is 1 to 10 (hexadecimal). The display will prompt "06" for the inter-cycle delay desired; this may be in the range of 0 to FF, and will produce delays of 0 to 2.55 seconds, respectively. The prompts "07" and "08" may be ignored, or used for extra correlation information as desired. When the "word entry" key is released after the "08" prompt, HDR will record the information read from the clock and keyed in, and the DEMO routine will then call DGTR to acquire the data requested.

When DGTR has finished recording the data on tape, it will return to DEMO, which then calls HDR again. This sequence will be repeated indefinitely until the machine is reset.

## V. SUMMARY AND CONCLUSIONS

MIDAS combines a high-speed multiplexed data acquisition system module, an incremental digital tape cassette recorder, and a second-generation microcomputer in a compact, lightweight package. Requiring only external electrical power, the system is capable of completely unattended operation, and may be programmed to operate in literally an infinite number of modes.

The results obtained using MIDAS to date have been encouraging, resulting in the production of three other hand-built systems. The prototype for a dedicated airborne system is in its initial stage. Used to monitor the fatigue-critical points in an airframe structure, the device will be extremely compact and, if put into production, may cost less than two thousand dollars, even using militarized parts.

# A Machine-Independent Floppy Disk Controller with Portable Software Support

Brian Johnson
The University of Texas at Dallas

Portia Isaacson
The Micro Store

## Abstract

This paper describes a flexible disk controller design based on a microprocessor. The controller provides a very simple time-independent interface to the host computer. The disk with controller can be readily connected to many different hosts because of the simplicity of the interface and the portable software resident in the controller microprocessor.

## Introduction

In this paper we describe a floppy disk controller designed for a computer science laboratory containing several kinds of computers. One objective of the controller design was to choose a single host/ controller interface suitable for all computers. In the computer science laboratory, student projects include the design and implementation of operating systems and file-management systems. Therefore, another objective of the disk controller was to present a logical interface to the host computer that did not preclude experimentation with file and directory structures. Since the host computer software would in many cases be in the development stage, it was desirable that the disk controller be able to detect as many errors as possible in the usage of the interface by the host computer. A machine-independent floppy disk controller having a RS-232C (1) standard interface (as shown in Figure 1) was implemented. The portability of the RS-232C interface approach over a direct-to-bus parallel interface is well known and was felt to be more important than the loss of speed suffered by choosing a serial over a parallel interface. More important to the portability, the disk controller is itself a microcomputer containing its own disk handler software. The controller provides significant functional capability while handling all the troublesome unique properties of the disk--such as interrupts and fixed-size sectors.

In addition to host-microcomputer independence, a number of secondary advantages are realized from the intelligent-controller approach to the disk interface.

1. The host microcomputer software requirements are reduced. A significant portion of the operating system necessary to support the disk has been moved into the controller microcomputer, thus freeing scarce memory and execution-time overhead in the host microcomputer. The controller microprocessor provides multiple open file capability, a full-sector buffer for each of up to four open files, very efficient sequential file access both forward and backward, random access to the character level, update-in-place, reading and writing the same file, diskette initialization, efficient logical sector spacing around a track, and disk error recovery by retrying operations.

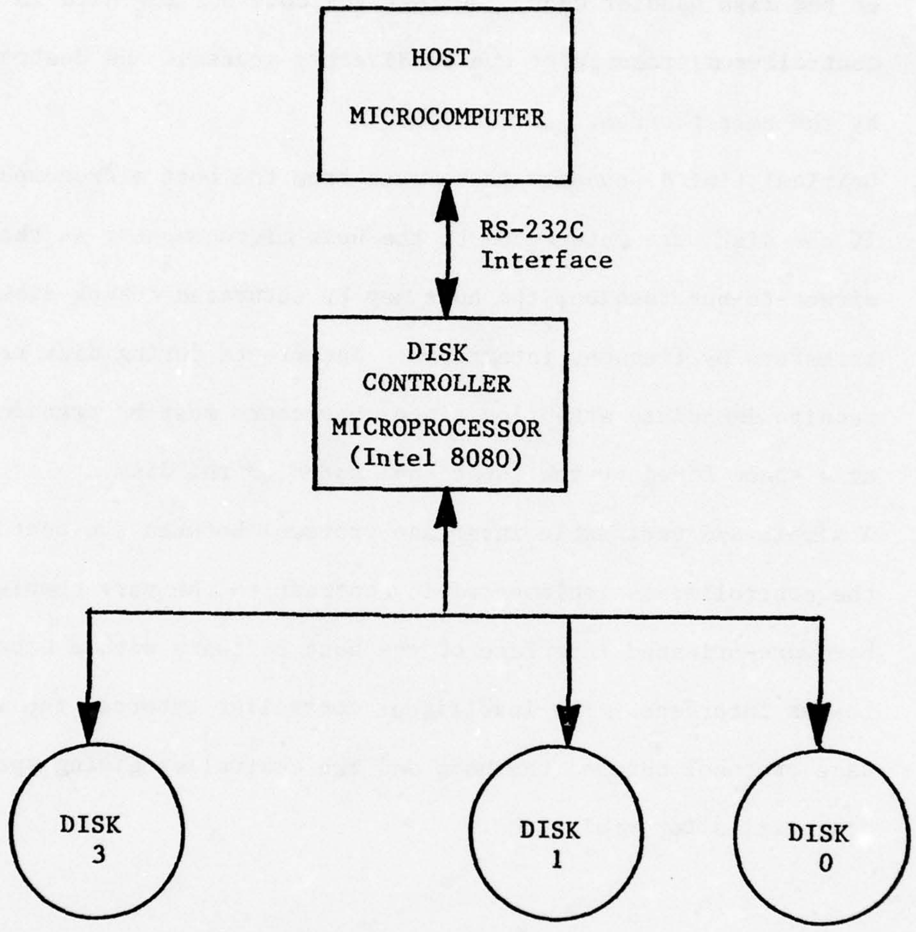


FIGURE 1. Intelligent Disk Interface

2. User program crashes are made less serious by the inaccessibility of the disk handler code. Neither the code nor the data in the controller microcomputer can be directly accessed and destroyed by the user program.
3. Critical time dependency is removed from the host microcomputer. If the disk were interfaced to the host microcomputer in the usual direct-to-bus fashion, the host may be saturated during disk transfers by frequent interrupts. Interrupts during disk transfers require immediate attention since characters must be transferred at a speed fixed by the rotational speed of the disk.
4. A simple and verifiable interface protocol between the host and the controller is implemented in contrast to the very complex hardware-oriented interface of the host software with a direct-to-bus interface. The intelligent controller enforces the interface protocol between the host and the controller giving specific diagnostics for violations.

## Host/Controller Communication

Commands and data are passed from the host microcomputer to the controller microcomputer. The controller returns data and status to the host. Data, commands, and status are all transmitted as serial asynchronous 8-bit quantities acrcss the RS-232C interface. Data is distinguished from commands and status using the most significant bit of each 8-bits, as shown in Figure 2. The advantage gained is that significantly fewer bytes need

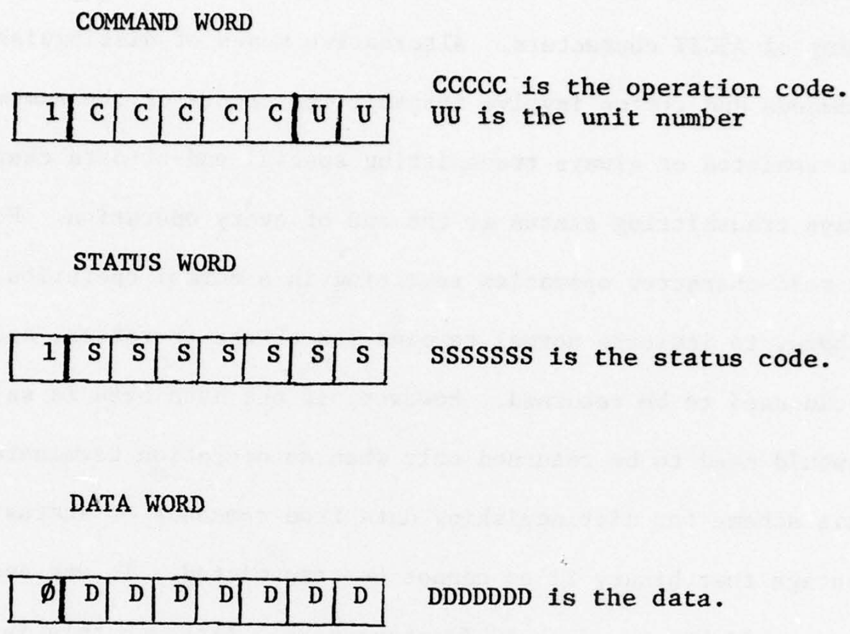


FIGURE 2. -- Command, status, and data words.

be transferred between the host microcomputer and the controller for files consisting of ASCII characters. Alternative means of distinguishing data from commands and status involve transmitting counts of the number of data bytes transmitted or always transmitting special end-of-data characters and always transmitting status at the end of every operation. For example, after a read-character operation resulting in a normal operation both a status byte, to indicate normal termination of the operation, and a data byte would need to be returned. However, if the data byte is self-identifying, status would need to be returned only when an operation terminated abnormally.

This scheme for distinguishing data from commands or status has the disadvantage that binary files cannot be transmitted. It was assumed that all files would be stored as ASCII characters. Although this is wasteful of space for object files, it has the advantage that all files can be printed and edited.

The software in the disk controller is a 1.25K byte program that performs the commands sent to it by the host computer, buffers data in sector-size blocks, transfers data to and from the physical disk, presents a simulated interface of four random or sequentially accessed virtual units to the host computer, and performs several other functions that can be inferred by the description of the controller commands and status words that follows.

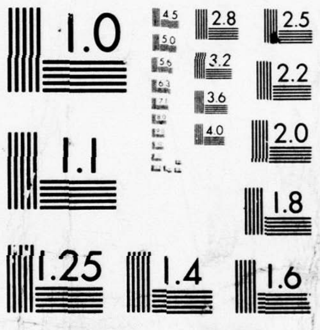
Fundamental to the communication between the host and the controller is the concept of a unit. The controller recognizes virtual units 0 through 3. Each virtual unit conceptually has a read/write head that is located at a "current" character. Physically the current character is located by four parameters associated with the virtual unit:

AD-A049 559 COLORADO STATE UNIV FORT COLLINS DEPT OF ELECTRICAL --ETC F/G 9/2
PROCEEDINGS OF THE ANNUAL ROCKY MOUNTAIN SYMPOSIUM ON MICROCOMP--ETC(U)
SEP 77 M ANDREWS, S MCCORMICK AFOSR-77-3367

UNCLASSIFIED AFOSR-TR-78-0017 NL

4 OF 4
AD
A049 559

END
DATE
FILMED
3 — 78
DDC


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

- the drive number which identifies which of up to four disk drives the current character is on,
- the track number which identifies which of 77 tracks on the disk the current character is in,
- the sector number which identifies which of 26 sectors on the track the character is in, and
- a character number which identifies which of the 128 characters within the sector is the current character.

Each disk is viewed as an array of characters formed by ordering the characters within each sector from 0 to 127, the sectors within a track from 0 to 25, and the tracks on a disk from 0 to 76. The array of characters on the disk are formed by laying the sectors within each track end-to-end and the tracks on the disk end-to-end, as shown in Figure 3. Viewing the disk as an array of characters simplifies sequential access of characters on a unit. Sequential access can be either forward or backward and is requested via the read/write-last-character and the read/write-next-character commands.

A read or write command to a virtual unit can be for the last character, the current character, or the next character. When the current character is read or written, the four parameters locating the current character do not change. However, when the last character or the next character are read or written, the track number, the sector number, and the character number are updated prior to the read or write operation so that the read/write head of the virtual unit is moved backward or forward, respectively, in the character array on the disk, as defined above.

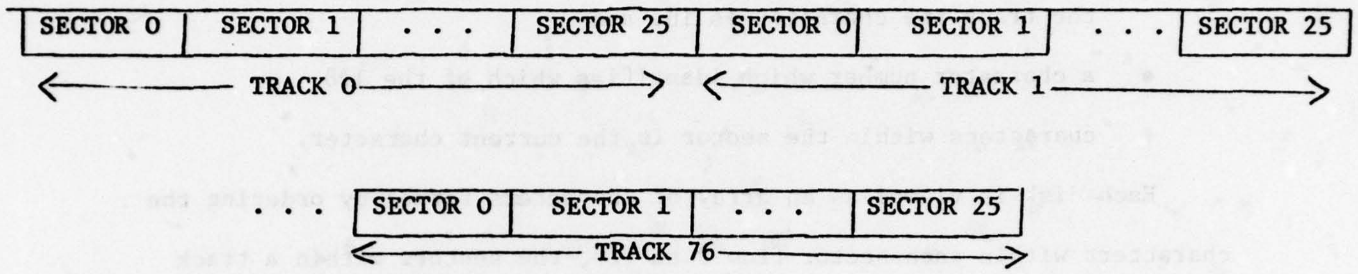


FIGURE 3. -- Array of characters on disk.

Random access to the character level is facilitated by commands which allow the disk number, the track number, the sector number, or the character number for a unit to be set at any time. By setting the location parameters prior to reading or writing the current character, data anywhere on any disk can be accessed. Once a location has been set, access can continue in a sequential manner, if desired.

Read and write commands on a unit can be mixed and; in fact, updates-in-place are readily accomplished. An update-in-place is a read operation followed by a write operation that replaces some or all the data read. This feature is very useful and is not commonly found in disk controller implementations. In this controller it is a natural result of the random access implementation.

Besides commands to read and write data and change the location of a virtual unit's read/write head, there are a number of other commands for such tasks as: closing units, initializing the controller, formatting a disk, and requesting status. A complete list is shown in Table 1.

Each virtual unit has an associated status word that is returned by the read-unit-status command. The status word contains four binary pieces of information: not-ready, write-protect, w-flag, and r-flag. The not-ready flag means the drive associated with the virtual unit does not contain a diskette or the disk door is not closed. The write-protect flag means the diskette has a write-protect seal. The w-flag indicates that the copy of the data in the buffer differs from the data on the disk; therefore, the sector must be written to the disk. The r-flag means that a copy of the disk sector has been placed in the buffer.

TABLE 1.

DISK CONTROLLER COMMANDS

u  is unit number in command,

d  is data word which must follow command, or data word returned as a result of command.

s  is status returned by controller

| Command, parameters | Controller Returns | Description |
|---|---|---|
| read drive, u | d | The drive number associated with unit u is returned as data to the host. |
| read track, u | d | The track number associated with unit u is returned as data to the host. |
| read sector, u | d | The sector number associated with unit u is returned as data to the host. |
| read byte, u | d | The character number associated with unit u is returned as data to the host. |
| set drive, u, d | s | The drive number associated with unit u is set to d if d is valid. S is "OK" or "invalid data." Any data in the buffer is written to the disk. |
| set track, u, d | s | The track number associated with unit u is set to d if d is valid. S is "OK" or "invalid data." Any data in the buffer is written to the disk. |
| set sector, u, d | s | The sector number associated with unit u is set to d if d is valid. S is "OK" or "invalid data." Any data in the buffer is written to the disk. |
| set byte, u, d | s | The character number associated with unit u is set to d if d is valid. S is "OK" or "invalid data." |

| Command, parameters | Controller Returns | Description |
|---|---|---|
| read previous character, u | d or s | The virtual unit u's location parameters are moved backward one character and that character is returned as d. If for any reason the character cannot be read, s is returned to indicate the problem. |
| read current character, u | d or s | The character defined by virtual unit u's location parameters is returned as d. If for any reason the character cannot be read, s is returned to indicate the problem. |
| read next character, u | d or s | The virtual unit u's location parameters are moved forward one character and that character is returned as d. If for any reason the character cannot be read, s is returned to indicate the problem. |
| write previous character, u, d | s | The virtual unit u's location parameters are moved backward one character and d is written at the new location. s is returned as "OK" or, if d cannot be written, s identifies the problem. |
| write current character u, d | s | The character defined by virtual unit u's location parameters is replaced by d. s is returned as "OK" or, if d cannot be written, s identifies the problem. |
| write next character, u, d | s | The virtual unit u's location parameters are moved forward one character and d is written at the new location. s is returned as "OK" or, if d cannot be written, s indicates the problem. |
| close, u | s | The virtual unit u is closed by writing any data in its buffer to the disk. s is returned as "OK" or, if the buffer cannot be written, s indicates the problem. |
| reset controller | | All units are initialized to disk 0, track 0, sector 0, and byte 0. The buffer associated with each unit is initialized to empty. All disk drive heads are returned to track 0. |
| restore disk | u | The disk drive head of virtual unit u is returned to track 0. |

| Command, parameters | Controller Returns | Description |
|---|---|---|
| read unit status | u | The status word associated with virtual unit u is returned. |
| format disk | u | The disk on virtual unit u is formatted. |
| format track | u | The track identified by virtual unit u's location parameters is formatted. |

After each command from the host microcomputer to the disk controller either data is returned, indicating "OK" status, or a status byte is returned which indicates either the normal termination of the command or the reason why the command could not be completed. Table 2 describes all status returns.

## Conclusion

The floppy disk controller described here has been implemented using the Intel 8080 microprocessor as the base of the controller microcomputer and the PerSci Flexible Disk Drive. This floppy disk subsystem has been connected to a SWTP-6800 microcomputer system for use as the primary mass storage for a personal computer operating system. The interface to the disk controller described here has been found to be easy to use and to have all the advantages discussed earlier. In addition, the partitioning of the usual operating system software which provides support for a disk into two pieces, the disk support in the disk controller microcomputer and the user interface in the host microcomputer, made the total system significantly simpler to implement. This simplicity was gained by isolating the troublesome disk-dependent considerations from the rest of the operating system.

Probably the major innovation of this approach is the portability of the device support software. The difficulties in creating portable software are well known. However, the inexpensive microprocessor has provided us with a perhaps-too-obvious solution to the portability problem. In order to create a portable software package, simply dedicate a computer to the software. What would at one time have been a ridiculous idea is now, because of the low cost of microprocessors, the most economical, reasonable, and straightforward solution.

TABLE 2.
STATUS RETURNS FROM DISK CONTROLLER

| Status | Description |
|---|---|
| OK | The previous operation was completed normally. |
| Drive not ready | The drive on which the previous operation was attempted does not contain a diskette or the disk door is not closed. |
| Write protected | The preceeding operation was an attempt to write on a diskette on which there was a write-protect seal. |
| Invalid command | The previous command code is not a command recognized by the disk controller or a data byte was received when a command was expected. |
| Invalid data | A command byte was received when a data byte was expected. |
| Seek error | The seek caused by the previous operation resulted in the disk head stopping at the wrong track. |
| Read/write error | The read or write caused by the previous operation terminated due to a media error on the diskette. |
| Disk underflow | The previous operation caused the location parameters to be decremented beyond the beginning of track 0. |
| Disk overflow | The previous operation caused the location parameters to be incremented beyond the end of track 76. |

## Bibliography

1. Electronic Industry Association, "Interface between data terminal equipment and data communication equipment employing serial binary data interchange," Specification Number RS-232C, August 1969.

# DISCRETE FOURIER TRANSFORM USING MICROCOMPUTER

Heng V. Te , Her-Daw Che

Department of Computer and Information Science

Moore School of Electrical Engineering

University of Pennsylvania

Philadelphia, PA 19174

## Abstract

The microprocessor provides a cost effective tool in that a designer can make use of the software flexibility to define and aid in hardware design. One example is the design of an FFT/DFT processor. Furthermore, as the dramatic decrease in both processor and memory costs and the increase of microprocessor speed continue, the software FFT/DFT package may become more attractive in some applications. A one-dimensional DFT algorithm implemented on an INTEL-8080 can provide, for 256 real inputs, one complex output every 260 milliseconds with six-bit precision. The entire system costs only six hundred dollars.

## Introduction

The basic discrete fourier transform (DFT) is defined by:

$$Ar = \sum_{k=0}^{N-1} Bk * e^{-2\pi rkj/N} \qquad \text{----------} \quad (1)$$

$$\text{where } j=(-1)^{1/2}$$

Ar is the r-th coefficient of the DFT and Bk denotes the k-th sample of the time series which consists of N samples. The procedure for calculating Ar is: rotate the vector Bk by the angle 2πrk/N and form the summation over k. This vector rotation is generally performed by four real multiplications and two additions. Golub [3] was able to accomplish the same result by using only three real multiplications and five additions. With minor modification to the trigonometric table, Buneman [4] suggested a way to do it with three real multiplications and three additions. Despain [1,2] came out with two very interesting approaches. One of them [2] is the famous CORDIC method which takes both real and imaginary inputs and employs only one multiplication. The other one [1] takes only real inputs, but uses a very simple technique and does not use multiplication at all. At one end of performance/cost tradeoff, if high throughput and precision are not required, the transform can even be done by an extremely inexpensive hardware device, say a microprocessor. A quick-and-dirty one-dimensional DFT routine has been programmed and tested on an INTELLEC-8 microcomputer at the Moore School of Electrical Engineering, University of Pennsylvania. The purpose of this paper is to examine the cost, performance, and precision of this package.

## Implementation

Equation (1) can be rewritten as:

$$Ar = \sum_{k=0}^{N-1} Bk*COS(2\pi rk/N) + j \sum_{k=0}^{N-1} Bk*SIN(2\pi rk/N) \text{ ---------- } (2)$$

If all the inputs are real and normalized within (-1, +1), Bk then can be represented by:

$$Bk = SIN(Zk) \text{ ------------------------ } (3)$$

where $-\pi/2 <= Zk <= \pi/2$. Hence,

$$Zk = SIN^{-1}(Bk) \text{ ------------------------ } (4).$$

Substituting (3) into (2), we obtain:

$$\begin{aligned} Ar &= \sum_{k=0}^{N-1} SIN(Zk)*COS(2\pi rk/N) + j \sum_{k=0}^{N-1} SIN(Zk)*SIN(2\pi rk/N) \\ &= 1/2 \sum_{k=0}^{N-1} [SIN(Zk+2\pi rk/N) + SIN(Zk-2\pi rk/N)] \\ &\quad + j/2 \sum_{k=0}^{N-1} [COS(Zk-2\pi rk/N) - COS(Zk+2\pi rk/N)] \text{ -------- } (5) \end{aligned}$$

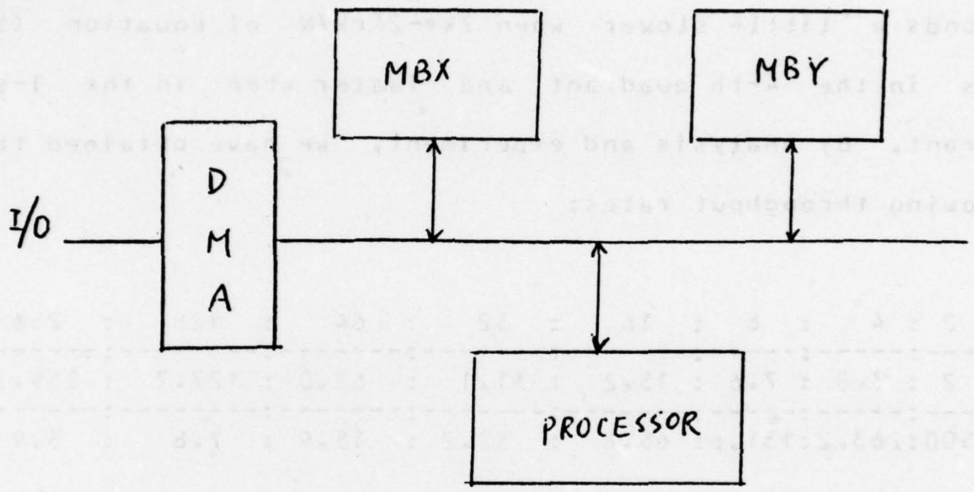
The last derivation results from the identities

SIN(a+b)+SIN(a-b)=2SIN(a)COS(b) and COS(a+b)-COS(a-b)= -2SIN(a)SIN(b). It is not unusual to restrict N (the number of input points) to be a power of two. If so, $2\pi rk/N$ can be obtained by shifting. It is now obvious that only additions/subtractions, shifts, and table look-ups are required, and multiplication is totally eliminated.

The program to calculate the DFT using this equation required 700 words (8 bits/word) and the associated tables needed 2,300 words. The hardware to perform this DFT consists of a processing unit based on the 8080 microprocessor, 3,000 words of read-only memory for the program and the tables, and 600 words of read-write memory for input buffers. In addition, we need the capability of Direct Memory Access (DMA). A typical commercial product fulfiling the above requirements is the INTEL SBC 80/20 single board computer. The whole system costs, including the power supply and the chasis, around six hundred dollars.

## Performance

Suppose I/O is done by DMA as shown below:



After steady state, while one set of data (say, MBX) is processed, the other previous calculated data (MBY) is output to I/O device. At the same time, another set of raw input is put in MBY via the DMA function. We can simply interchange the roles of MBX and MBY when one set of data has been completely processed. In this way, the I/O time is negligible compared to the DFT processing time. Since this is a DFT algorithm, each output point is produced successively. From equation (5), we know that the processing time for each Ar does not depend on that for other outputs. Performance is therefore greatly improved if multiple processor are employed. For instance, N microprocessors can be used to generate simultaneously N outputs.

The throughput rate depends almost linearly on the number of inputs (N). Due to our program structure, the program responds a little slower when $Zk+-2\pi rk/N$ of equation (5) falls in the 4-th quadrant and faster when in the 1-st quadrant. By analysis and experiment, we have obtained the following throughput rates:

| N | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|---|---|
| T | 2 | 3.8 | 7.6 | 15.2 | 31.1 | 63.0 | 127.7 | 259.2 |
| B | 500 | 263.2 | 131.6 | 65.8 | 32.2 | 15.9 | 7.8 | 3.9 |

where N is the number of inputs

T is the time for producing one successive output (milliseconds)

B is the processor bandwidth (hertz)

Obviously, the maximum sampling rate is very low. For 256 real input points, our package generates one output point every 259.2 milliseconds, or produces 256 output points in approximately 64 seconds. Thus our package constraints us to very low frequency applications.

## Precision

The input and output are quantized in 256 levels using an eight-bit 2's complement representation. Because the arguments (2k and $2\pi rk/N$) of the SIN/COS functions in eq. (5) may vary from $-\pi/2$ to $2\pi$ (6.2832), we need 8 bits to indicate fractional part of the argument, and a few bits to represent integer part. For simplicity, we use another 8 bits for integer part. The partial sums of SIN and COS in equation 5 are also kept in two 16-bit storages in order to allow a maximum number (256) of input points without overflow on accumulating the partial sum. Internal operations such as additions/substractions in eq. (5) are therefore handled in double precision. The result, Ar, is normalized ( i.e. divided by N ) before it is output.

We have run through test samples on both a large computer (Spectra 70/46), under the HARM Fortran subroutine yielding 32-bit precision outputs, and our quick-and-dirty package on INTELLEC-8. The HARM Fortran subroutine which performs Discrete Fourier Transforms on a complex three dimensional array is in the IBM Scientific Subroutine Package (SSP). Inputs to these two programs are ranged from -128 to 127, a representation from -1 to +1 in 256 level. The significant 8 bits of both outputs are compared. They checked correctly to first six bits.

## Conclusion

For real-time applications, we can buy a hardwired FFT processor which can perform 1024 complex inputs within 5 milliseconds. Such a processor costs around twenty to thirty thousand dollars. In some off-line applications, such as CRT synthesis or enhancing photographs and TV images, sometimes we do not need high throughput and accuracy. For such applications, this low cost DFT package may become attractive. The maximum input sampling rate for continous processing is around 4 hz for 256 points at a time. At the present pace of progress in microprocessor, the speed of the DFT will quickly increase. Furthermore, instead of the INTEL 8080 (MOS) chip, if we use bipolar (TTL) or even ECL processors, the speed gain will be 20 to 100 times higher. In order to get even higher performance, pipelined or parallel-processing microprocessor architectures may be used.

## Acknowledgement

The authors thank Dr. Robert C. Chen for reviewing the paper.

Reference

1. Despain,A.M; 'Digital Fourier Transform and Digital Filters Without Multiplications.' IEEE Communication Technology Conference, pp. 114-118 1975

2. Despain,A.M; 'Fourier Transform Computers using CORDIC iterations.' IEFE Trans. on Computer, pp. 993-1001 Oct. 1974

3. Singleton,R.C; 'An Algorithm for Computing the Mixed Radix Fast Fourier Transform.' IEEE Trans. Audio Electroacoust; pp. 45-55 June 1967

4. Buneman,O; 'Inversion of the Helmholtz Operator for Slab Geometry.' Inst. for plasma research, Stanford Univ., Stanford Calif., SUIPR REP. 467 p. 5, April 1972

Submitted to the First Annual Rocky Mountain Symposium on Microcomputers
August 31 - September 2, 1977

# A MICROPROCESSOR-BASED TERMINAL SWITCHING SYSTEM

Shuzo Yajima, Yahiko Kambayashi, Hiroyuki Ogino and Narao Nakatsu

Department of Information Science,
Kyoto University, Kyoto, JAPAN

Mailing Address: Yahiko Kambayashi
Department of Information Science
Kyoto University
Sakyo, Kyoto 606 JAPAN

# A MICROPROCESSOR-BASED TERMINAL SWITCHING SYSTEM

Shuzo Yajima, Yahiko Kambayashi, Hiroyuki Ogino and Narao Nakatsu

Department of Information Science,
Kyoto University, Kyoto, JAPAN

Abstract

Towards computer-based research and education in the laboratory, we have developed an optically linked laboratory oriented computer network LABOLINK. It consists of a minicomputer PDP-11/40 in our laboratory linking with a medium scale computer HITAC 8350 of the Department of Information Science and a large scale computer FACOM 230-75 of the Data Processing Center, Kyoto University. We have developed a microprocessor-based terminal switching system in order to use any computer in the LABOLINK from any terminal located within our laboratory. Tedious modification of the operating system of the PDP-11/40 was avoided through the introduction of a microprocessor. The main objectives of the terminal switching system are as follows.
(1) Any remote terminal can be used as the console terminal of the PDP-11/40.
(2) Any computers in LABOLINK can be accessed from any terminals.
(3) Terminal-specification differences are compensated.
(4) Terminal users can communicate each other by a conversational mode.
(5) Several terminals can be controlled simultaneously by a high-level language program (FORTRAN or BASIC) of the PDP-11/40 without modifying input/output statements.
(6) New terminals can be installed without adding device drivers.
(7) If more than one computer is connected to the terminal switching system, communication between computers can be realized.
By using Multi-User BASIC and foreground-background capability of the PDP-11's RT-11 monitor, simultaneous use of terminals is possible. For example, two users can access the HITAC 8350 and the FACOM 230-75 independently as foreground Multi-User BASIC jobs while another user edits programs as a background job, since network control programs of the PDP-11/40 are written in Multi-User BASIC (for this purpose, original PDP's Multi-User BASIC is extended).
The microprocessor-based terminal switching system discussed in this paper can be used in other computer systems with little modification.

## 1. INTRODUCTION

We have developed an optically linked laboratory oriented computer network LABOLINK to be used in computer-based research and education in an environment of a university laboratory.
The LABOLINK consists of a minicomputer (PDP-11/40) in our laboratory linking with a medium scale computer (HITAC 8350) of the Department of Information Science and a large scale computer (FACOM 230-75) (to be replaced by a FACOM M190 which is equivalent to the Amdahl computer) of the Data Processing Center, Kyoto University.
It is desirable that any computer in LABOLINK can be used simultaneously by any terminal distributed within our laboratory. Usually this is achieved by a modification of the operating system which is known as a tedious work. We have developed a microprocessor-based system for this purpose, since by this approach we only need to treat a microprocessor system instead of a minicomputer and the system also can be used in other computer system with different operating system with little modification. Multi-User BASIC of the PDP-11/40 is also modified to handle interrupt signals and device status registers, which is used to write network control programs as well as application programs such as a manuscript preparation system. Using the microprocessor-based terminal switching system, extended Multi-User BASIC and RT-11 monitor with foreground-background capability, simultaneous realization of connections between computers and terminals is possible. For example, two users can use the HITAC 8350 and the FACOM 230-75 independently as foreground Multi-User BASIC jobs while another user edits his paper by the manuscript preparation system as a background job.

The main objectives of the microprocessor-based terminal switching system are summalized as follows.
(1) Any terminal can be used as the console terminal for the PDP-11/40. That is, from any terminal, monitors (DOS or RT-11), language processors (FORTRAN or BASIC) and utility programs can be loaded.
(2) It is possible to control the microprocessor by programs written in a high-level language such as Extended Multi-User BASIC in order to simplify the program development.
(3) Any terminal can be used as a terminal device of any program run on the PDP-11/40. Thus any computer of LABOLINK can be accessed from any terminal. Simultaneous usage of terminals is possible since network control programs are written in Extended Multi-User BASIC.
(4) Terminal-specification differences are compensated. Examples are as follows. Some terminals only accept capital letters while others accept both upper and lower case letters. Some terminals realize a composite operation of carriage return (CR) and line feed (LF) when they receive a single control code LF or CR.
(5) Terminal users can communicate each other by a conversational mode. This function is useful when message transmission is necessary between terminals, for example, "when do you finish your work?", "how can I use your program?" etc.
(6) Simultaneous control of several terminals by a high-level language program is possible. A game program which uses several terminals is easily implemented. This function is especially important in cases such as the manuscript preparation system which requires two typewriters , one high resolution printer (Diablo HyType I) for final manuscript printing and the other for control and error messages.
(7) New terminals can be installed without adding device drivers and they are also controlled by previously developed software.

Further functions to be added to the system are as follows.
(8) Communication between computers can be realized when more than one computer is connected to the terminal switching system. Communication between two independent programs running simultaneously on the PDP-11/40 is also possible.
(9) Both foreground and background monitor messages are printed on the console terminal, which are distinguished by the heading "F" or "R". This two kinds of messages should be distributed to two different terminals.
(10) Further compensation of terminal specification differences is necessary. For example, code conversion between ASCII and EBCDIC is necessary when we have to use EBCDIC terminals. In order to achieve these objectives, the terminal switching system has four modes for each terminal and computer interface.

## 2. OUTLINE OF THE LABOLINK

Although the LABOLINK is desined as a typical laboratory oriented computer network, it has several characteristic features in both its software and hardware. Configuration of LABOLINK is shown in Fig.1. A minicomputer PDP-11/40 located at our laboratory is linked with a medium scale computer HITAC 8350 of the Department of Information Science and a large scale computer FACOM 230-75 of the Data Processing Center, Kyoto University. An optical fiber cable with line speed of 1 Mbps links the PDP with the HITAC. We have linked the PDP with the FACOM, which is to be a host of Japanese inter-university computer network partly under development, by a 1200 bps TSS line. An optical fiber cable was installed to establish broad band communication lines among research rooms. The optical cable itself can transmit signals at 100 Mbps, but is currently at 4 Mbps due to the restriction of photo-electric converters. A single line multi-connection (SLMC for short) is developed to use a broad band optical fiber line as many communication lines. The current version of SLMC can handle at most 16 communication channels. It automatically determines the number of active channels and varies its speed according to it. That is, it can be used as a very high speed line as well as several lower speed lines according to the number of active channels. SLMC employs high efficient self synchronizing transmission code, and its maximum possible transmission rate is 4 Mbps. As a suitable model of computer communication interfaces a new class of automata called two-I/O-pair automata is introduced. It has two independent input-output pairs. As in the case of actual interfaces, a machine connected one input-output pair is unable to know the input-output sequences realized at the other input-output pair directly, although the

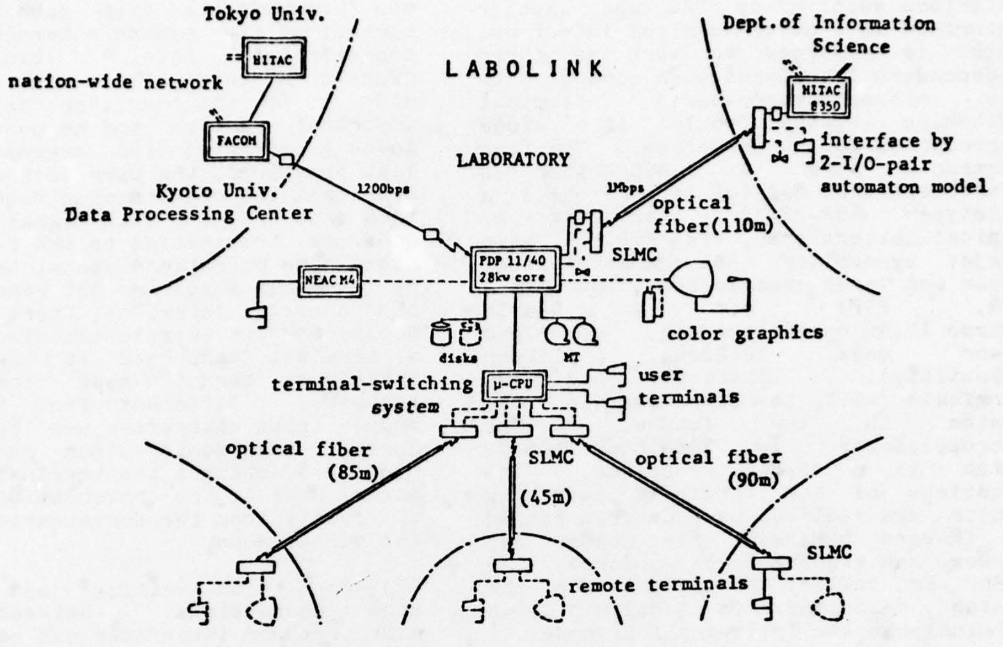


Fig.1 Laboratory computer network LABOLINK

output of the former pair may contain some information about the latter. This property causes specific problems which need not be considered in conventional automata. Using this model automata theoretic approach for designing computer interfaces is realized(for example, reduction of communication symbols, reduction of internal states, increase of reliability can be achieved). We have designed computer interfaces between the PDP and the HITAC by this approach, which have higher error detecting capability. Communication software of the HITAC is written in assembly language while communication software of the PDP is written in Extended Multi-User BASIC, which is a modification of RT-11 Multi-User BASIC by adding commands to handle device status registers and device interrupt signals. Since RT-11 Multi-User BASIC can be used by up to 8 terminals, simultaneous use of the FACOM and the HITAC is possible as well as other BASIC programs. File transfer between the PDP and the HITAC can be done by a simple command string according to the RT-11 monitor standard format. As the HITAC and the PDP use EBCDIC and ASCII codes, respectively, code conversion is necessary. In order to reduce program size for the network control program, we have developed a hardware code converter. Two memory addresses are assigned to EBCDIC-to-ASCII and ASCII-to-EBCDIC converters both of which are realized by LSI code converters. To make a code conversion, we just need to store a word and then read a word at the same address. Code conversion is completed before the read operation.

A microprocessor-based terminal switching system is introduced to access computers in the LABOLINK from terminals distributed in our laboratory. We have currently four terminals; a DEC Writer, a Teletype ASR-33, a Casio Inkjet typewriter and a Diablo HyType I.

## 3. CONFIGURATION OF MICROPROCESSOR - BASED TERMINAL SWITCHING SYSTEM

Fig.2 shows a configuration of the microprocessor-based terminal switching system. Two computer interfaces for the PDP-11/40 and four terminal interfaces are currently connected to the system. All these interfaces will be called microprocessor interfaces. One computer interface is a console interface supplied by DEC and another computer interface is a console interface supplied by DEC and another interface is a multi-terminal interface. This is designed to work as three independent interfaces when connected to the microprocessor-based terminal switching system while it alone corresponds to one interface. The four terminals are a DEC Writer (30 characters/sec, capital letters only), a Teletype ASR-33(10 characters/sec, capital letters only, PTR, PTP), a Casio inkjet typewriter (30 characters/sec, upper and lower case letters, low noise, PTR, PTP) and a Diablo HyType I (30 characters/sec, upper and lower case letters, plotter capability). Character display terminals will be connected to the system in the future. The microprocessor is TOSHIBA's TLCS-12 which is a 12-bit processor. The functions of the terminal switching system are realized by a program stored in 2k-word ROM(read only memory) and 1k-word RAM (random access memory). In order to realize the objectives of the system described in Section 1 each terminal has the following four mode.

(1) Normal mode : Signals received from one terminal or computer interface are transmitted to another terminal or computer interface according to the connection registers of the microprocessor. Each terminal or computer interface is at the normal mode when its power is on. Before it become off each terminal user must type %E in order to erase all connections which was set up by this microprocessor interface.

(2) Conversation mode : Terminal users can communicate with each other by operating the sender's terminal in the conversation mode. Usually the receiver's terminal is in the normal mode. If the receiver is printing important results and he does not want to be interrupted (for example, program list printing), the user of the terminal can mask the conversation request. In this case only the BELL signal of ASCII code is transmitted to the receiver. Since some operations cannot be realized by terminals which are not connected to the console interface, there is more demand for the console interface. Thus a terminal connected to the console interface cannot mask conversation requests. Different from the normal mode, input characters are echo-backed during the conversation mode. The command %C changes the terminal from the normal mode to the conversation mode and %N resets from the conversation mode to the normal mode.

(3) Connection setting and display mode : Connections between two microprcessor interfaces can be set up by any terminal or computer interface. So before the terminal power is off (or before the end of the program which set up the connection) %E must be sent to the processor in order to erase connections determined by the terminal or the computer interface. In order to control several terminals simultaneously by a high-level language program, two kinds of connections called real connections and virtual connections are used. If a terminal or a computer

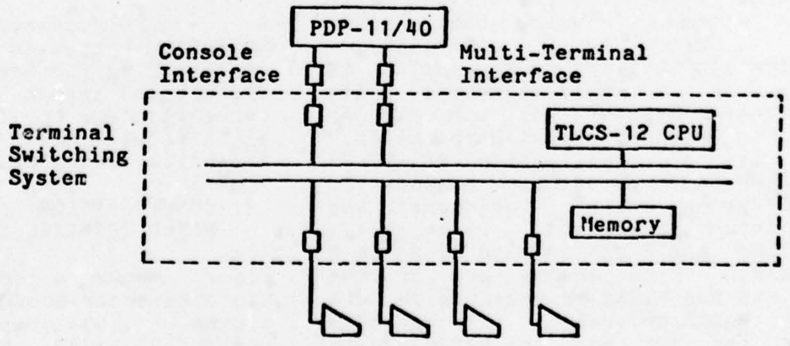


Fig.2 A microprocessor-based terminal switching system

interface A has a real connection to another terminal or computer interface B, signals given from A are transmitted to B. If the connection is virtual no such signal transmission occurs, but the other microprocessor interface cannot make a connection (real or virtual) to B. When we want to control several terminals simultaneously by a high-level language program, first we set up virtual connections from the computer interface used by the program to the terminals to be controlled. Every time before controlling one of these terminals the mode of the computer interface must become the connection setting and display mode and one of virtual connections should be changed to a real connection. Display command D is used to print the current status of the connection. Two commands %A and %N change the microprocessor interface mode from the normal mode to the connection setting and display mode and vice versa, respectively.

(4) Microprocessor mode : Contents of the microprocessor memory can be altered by inputs given from a microprocessor interface. This mode can be used by computer interfaces to load programs to the microprocessor. A terminal interface can be at this mode only when no other terminal and computer system use the terminal switching system. Command %M changes from the normal mode to the microprocessor mode and %N resets to the normal mode.

Commands related to the mode transition are summalized as Fig.3.
In order to set up a real connection between one terminal and one computer interface the following procedure can be used.

(a) Input %A in order to change the mode of the terminal to the connection setting and display mode.
(b) Input a two-way connection command which will establish a two-way connection between the terminal and one computer interface. If it fails (i.e. the computer intrface is occupied), error message is printed, otherwise go to (e).
(c) Input a display command in order to know which terminal is using the computer interface.
(d) Change the mode of the terminal into the conversation mode in order to communicate with the computer interface user. Wait until the other user finishes and go to (a).
(e) Input %N command to change the mode of the terminal normal. Start to use the terminal.
(f) After finishing the job input %E command for the termination.

To send commands from a computer to the microprocessor is realized by PRINT or WRITE statement of high-level languages. Since each command must be followed by a carriage return and line feed, each command should be written in an independent I/O statement like

```
      PRINT     1000
 1000 FORMAT  (3H %M,/)
```

For the control of terminals T1,...,Tn simultaneously by a high-level language program the following procedure is used.
(a) Create virtual connections to the terminals T1,...,Tn from the computer interface which is used by the program.
(b) When it is necessary to transmit signals to and from terminals, change the corresponding virtual connections to real.
It is possible to print out identical messages to more than one terminal by a single PRINT statement.

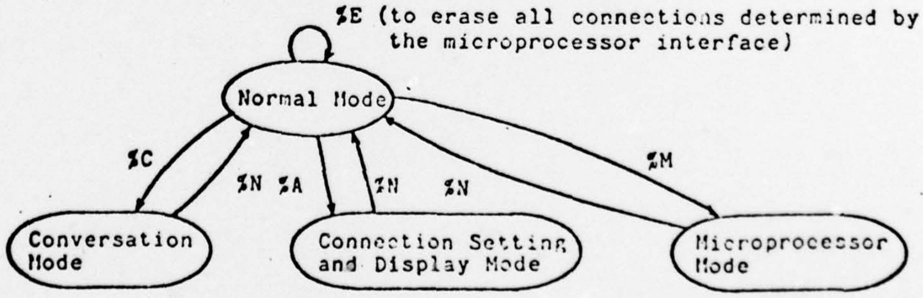

%E (to erase all connections determined by the microprocessor interface)
Normal Mode
%C
%N %A
%N
%N
%M
Conversation Mode
Connection Setting and Display Mode
Microprocessor Mode

Fig.3 Commands for mode transition

## 4. COMMANDS FOR EACH MODE

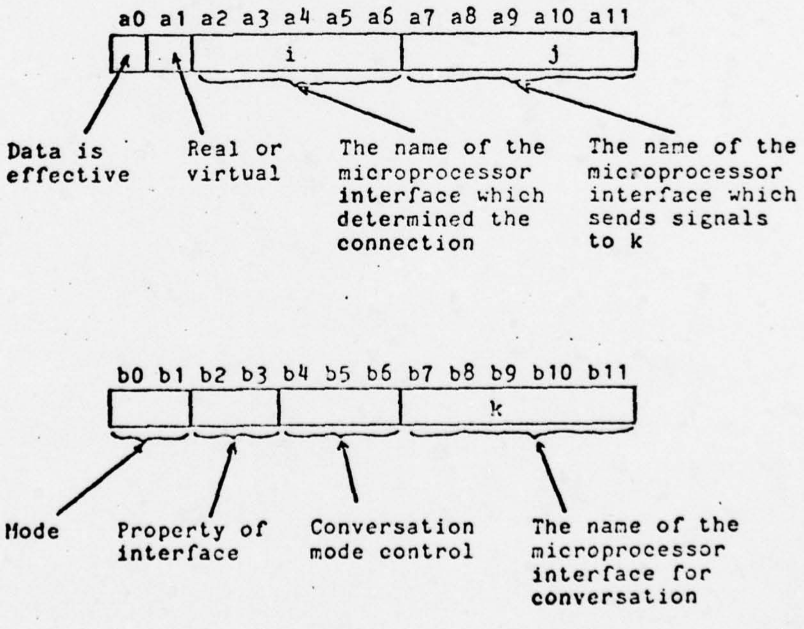
The following commands are used at the connection setting and display mode, where i and j are octal numbers (0≤i,j≤31) corresponding to the names of microprocessor interfaces. Let k be the name of the microprocessor interface which gives the following commands.

| | |
|---|---|
| O i j | : Create a one-way real connection from i to j. |
| O j | : Create a one-way real connection from k to j. |
| P i j | : Erase a one-way real connection from i to j. |
| P j | : Erase a one-way real connection from k to j. |
| T i j | : Create a two-way real connection between i and j. |
| T j | : Create a two-way real connection between k and j. |
| U i j | : Erase a two-way real connection between i and j. |
| U j | : Erase a two-way real connection between k and j. |
| R j | : Create a virtual connection between k and j. |
| S j | : Erase a virtual connection between k and j. |
| N i | : Erase all connections (one-way, two-way, virtual) to and from i. |
| D i | : Display all numbers of microprocessor interfaces which have connections with i. |
| D | : Display all connections. |

The following commands are used at the conversation mode.

| | |
|---|---|
| C j | : Conversational interrupt for interface j. |
| V | : Inhibition of any conversation request for k (it is effective when it is not connected to the PDP-11/40 console interface). |
| W | : Permission of conversation for k (it is used after V is valid). |

In order to treat mistakes occured during command and conversation message input, the following commands are



a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11
i
j
Data is effective
Real or virtual
The name of the microprocessor interface which determined the connection
The name of the microprocessor interface which sends signals to k
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11
k
Mode
Property of interface
Conversation mode control
The name of the microprocessor interface for conversation

Fig.4 Connection registers for microprocessor interface k

## 4. COMMANDS FOR EACH MODE

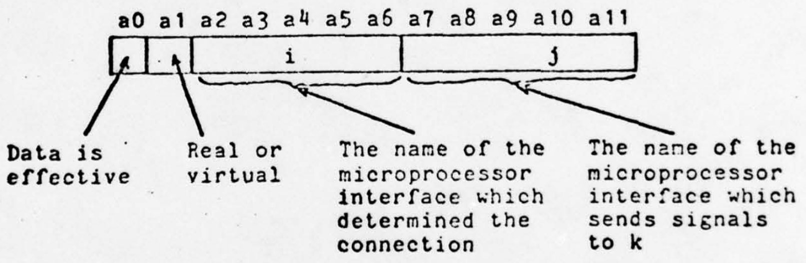
The following commands are used at the connection setting and display mode, where i and j are octal numbers ($0 \leq i, j \leq 31$) corresponding to the names of microprocessor interfaces. Let k be the name of the microprocessor interface which gives the following commands.

| | |
|---|---|
| **O i j** | : Create a one-way real connection from i to j. |
| **O j** | : Create a one-way real connection from k to j. |
| **P i j** | : Erase a one-way real connection from i to j. |
| **P j** | : Erase a one-way real connection from k to j. |
| **T i j** | : Create a two-way real connection between i and j. |
| **T j** | : Create a two-way real connection between k and j. |
| **U i j** | : Erase a two-way real connection between i and j. |
| **U j** | : Erase a two-way real connection between k and j. |
| **R j** | : Create a virtual connection between k and j. |
| **S j** | : Erase a virtual connection between k and j. |
| **N i** | : Erase all connections (one-way, two-way, virtual) to and from i. |
| **D i** | : Display all numbers of microprocessor interfaces which have connections with i. |
| **D** | : Display all connections. |

The following commands are used at the conversation mode.

| | |
|---|---|
| **C j** | : Conversational interrupt for interface j. |
| **V** | : Inhibition of any conversation request for k (it is effective when it is not connected to the PDP-11/40 console interface). |
| **W** | : Permission of conversation for k (it is used after V is valid). |

In order to treat mistakes occured during command and conversation message input, the following commands are


a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11
i
j
Data is effective
Real or virtual
The name of the microprocessor interface which determined the connection
The name of the microprocessor interface which sends signals to k

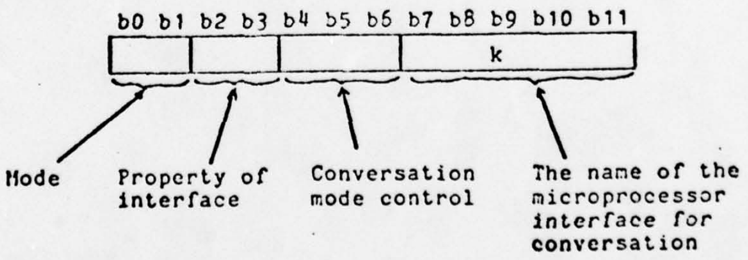

b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11
k
Mode
Property of interface
Conversation mode control
The name of the microprocessor interface for conversation

Fig.4 Connection registers for microprocessor interface k

prepared.

| | | |
|---|---|---|
| BS(Back Space) | : | One character correction. |
| # | : | Erase all characters in the current line. |

Two kinds of internal registers are prepared for each microprocessor interface to realize the functions of the terminal switching system. These are called connection registers and command registers. Bit assignments for connection registers are shown in Fig.4.

## 5. APPLICATIONS

Typical applications of the system are as follows:

(1) Simultaneous use of computers in the LABOLINK from terminals located in the laboratory: Since the RT-11 monitor has a foreground and background capability, one backgroung job and up to eight foreground Multi-User BASIC jobs can be executed simultaneously.

For the simultaneous use of computers, communication programs are written in Multi-User BASIC by adding statements for communication control.

These statements are chiefly as follows: i) a statement for moving the contents of a specified address, including I/O registers, to a variable in BASIC or vice versa, ii) a statement for assembling a data block transmitted from the FACOM and putting it into a BASIC string variable, iii) two statements for message exchange with the HITAC and a statement for setting up an interrupt vector address. For example, WORD(A,B) assigns the content whose address is the first argument, to the second argument. TXTR(A$,C) receives a data block from the FACOM and assigns it to a string variable A$, where C=1 shows that no response is received during the predetermined time period.

The following programs are prepared in computers in the LABOLINK.

PDP: Small programs for research aid. Game playing programs. A manuscript preparation system.

HITAC: A PL/I cross assembler for the terminal switching microprocessor. Programs for data base research. On-line information retrieval system of papers in computer science.

FACOM: Programs for automata theory, logic design and combinatorial theory which can be used from LABOLINK terminals.

A typical simultaneous usage of the system is as follows. One user prepares a FORTRAN program as a background job of the PDP, while one or two other users use the PDP, the FACOM or the HITAC under foreground Multi-User BASIC programs.

(2) A manuscript preparation system:

As is the case of most application programs of the LABOLINK which are written in BASIC or FORTRAN, high-level language control of a high resolution typewriter Diablo HyType I is possible. It is achieved by developing a control hardware which can be connected to conventional typewriter interfaces of computers. Its control-and-status register can be altered by ASCII control characters or special combinations of printable characters. Programs of the manuscript preparation system and simple picture drawing system are written in Multi-User BASIC. This paper is prepared by the manuscript preparation system. An example of a figure produced by a simple picture drawing system written in BASIC

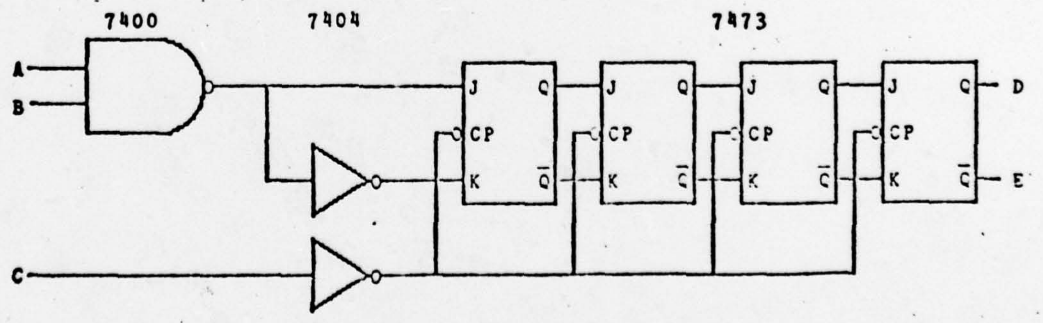


Fig.5 An example of a figure produced by a simple picture drawing system

is shown in Fig.5.

Usually the PDP-11 single-user on-line TEXT EDITOR is used to store the contents of a manuscript on DEC tapes. If simultaneous usage is required, an editor with restricted capability written in Multi-User BASIC can be used. The system uses two terminals, one for output printing and the other for control messages. Usage of two terminals is realized by the multi-terminal control capability of the terminal switching system.

## 6. CONCLUDING REMARKS

A microprocessor-based terminal switching system seems to be a reasonable method for realizing a minicomputer with many remote terminals. However, for the simplicity of the system, control for resource sharing is left to the users. It will not be a serious problem if only one computer is connected to the system. We are planning to expand the LABOLINK by introducing another computer in our laboratory. Both the PDP and the new computer will be connected to the terminal switching system and thus further extension of the terminal switching system will be required.

## ACKNOWLEDGEMENT

The authors would like to thank the members of Yajima laboratory for their cooperation during the development of LABOLINK and the terminal switching system. Especially they are indebted to Mr.Susumu Yoshida, Mr.Hiromi Hiraishi, Mr.Kazuo Iwama, Mr.Itaru Okuno, Mr.Norio Tomii and Mr.Hideyuki Hara for their assistance on this work. This work is supported by the Science Foundation Grant of the Ministry of Education, Science and Culture of Japan.

## REFERENCE

[1] S.Yajima et.al, "Optically linked laboratory computer network LABOLINK", Proceedings of the Tenth Hawaii *International Conference on System* Sciences, pp.1-4, January 1977.